DOT

MS™-BASIC LANGUAGE MANUAL

MS™-BASIC LANGUAGE MANUAL

0008-14662
© Copyright 1979, 1981
Microsoft Corporation

Computer Devices, Inc.
25 North Avenue
Burlington, MA 01803
TEL: (617) 273-1550

The information in this document is subject to change without notice and should not be construed as a commitment by Computer Devices, Inc. and in the case of software authored by third parties, should not be construed as a commitment by such third party.

Software that may be described in this document is furnished under the Software Program License Agreement which is provided in its entirety below.

Computer Devices, Inc. and the author of software in the case of software authored by such third party, assumes no responsibility for the use or reliability of its software or for any errors that may appear in this document.

COMPUTER DEVICES, INC.

PROGRAM LICENSE AGREEMENT

COMPUTER DEVICES, INC. ("LICENSOR") PROVIDES THE ENCLOSED PROGRAM ("THE PROGRAM") AND LICENSES ITS USE ONLY ON THE FOLLOWING TERMS AND CONDITIONS. YOU SHOULD CAREFULLY READ THE TERMS AND CONDITIONS BEFORE OPENING THIS PACKAGE. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE, UNOPENED, TO LICENSOR OR ITS AUTHORIZED DISTRIBUTOR, AND YOUR MONEY WILL BE REFUNDED.

1. COPYRIGHT.

The Program and its documentation are copyrighted by Licensor or its supplier. You may not use, copy, modify, or transfer the Program, or any copy, modification, or merged portion of the Program or documentation except as expressly authorized in this Agreement.

2. LICENSE.

You may use the Program only on a single machine. You may transfer the Program and the License from one machine to another, provided that you do not use the Program on more than one machine at a time. You may not make the Program available through a network of machines unless expressly authorized by the Licensor in writing.

3. BACKUP COPIES.

If the Program is not marked "copy protected" you may make one machine readable copy of the Program for backup purposes in support of your use of the Program on the single machine. If the Program is marked "copy protected" and a backup copy is not included in this package, you may obtain one by completing the enclosed user registration card and returning it to the Licensor with a request for a backup copy of the Program. You may not copy the Program or any of its documentation in printed form.

4. MODIFICATION.

You may not modify the Program or merge it into another program, except as may be expressly authorized in the user manual for the Program. Any portion of the Program that is merged into another program will continue to be subject to the terms and conditions of this Agreement. You must, as a condition of the License, reproduce and include the copyright notice on any copy, modification, or merged portion of the Program.

5. ASSIGNMENT.

You may transfer the program and assign your rights under this agreement to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the Program, you must at the same time either transfer all copies thereof in printed or machine readable form to the same party or destroy any copies not transferred. If you transfer the Program, your license is automatically terminated.

6. TERM.

The License granted in this Agreement is effective until terminated. You may terminate it at any time by destroying the Program together with all copies, modifications, and merged portions and notifying Licensor. The License granted in this Agreement will also terminate if you fail to comply with any of the terms and conditions of this Agreement. Upon such termination, you must destroy the Program together with all copies, modifications, and merged portions thereof.

7. LIMITED WARRANTY.

Licensor warrants only (a) that the media on which the Program is furnished will be free from defects in materials and workmanship under normal use for a period of 90 days from the date of delivery to you as evidenced by a copy of your receipt.

ALL SOFTWARE PRODUCTS LICENSED BY COMPUTER DEVICES, INC. ARE PROVIDED AS IS WITHOUT ANY WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED.

YOU ASSUME FULL RESPONSIBILITY FOR THE SELECTION OF THE PROGRAM TO ACHIEVE YOUR INTENDED PURPOSES, FOR THE PROPER INSTALLATION AND USE OF THE PROGRAM AND FOR VERIFYING THE RESULTS OBTAINED FROM USE OF THE PROGRAM. LICENSOR DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE PROGRAM WILL MEET YOUR REQUIREMENTS, THAT THE PROGRAM IS FIT FOR ANY PARTICULAR PURPOSE, OR THAT THE OPERATION OF THE PROGRAM WILL BE UNINTERRUPTED OR ERROR FREE.

Should you discover a defect in the recording medium within the 90 day warranty period, return the defective medium to Licensor at the address indicated below. Include a copy of your receipt or other proof that the 90 day warranty period has not expired. Licensor will, upon verfication of the defect either replace the defective copy or refund the amount you paid.

THIS WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, WHETHER ORAL OR WRITTEN, EXPRESS OR IMPLIED. ANY IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO 90 DAYS FROM THE DATE OF DELIVERY TO YOU.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. This limited warranty gives you specific legal rights and you may also have other rights which vary from state to state.

8. LIMITATION OF REMEDIES.

YOUR EXCLUSIVE REMEDY AND LICENSOR'S ENTIRE LIABILITY SHALL BE AS SET FORTH ABOVE. IN NO EVENT WILL LICENSOR BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM OR THE BREACH OF ANY EXPRESS OR IMPLIED WARRANTY EVEN IF LICENSOR OR ITS DEALER OR DISTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

9. GENERAL

You may not sublicense, assign, or transfer the License or the Programs except as expressly provided in this Agreement. Any attempt to do so shall void the license.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREEE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND LICENSOR AND SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT OR ANY OTHER COMMUNICATIONS BETWEEN YOU AND LICENSOR RELATING TO THE USE OF THE PROGRAM.

If you have any questions regarding this Agreement, you may contact Licensor by writing or calling Licensor at the following address:

Computer Devices, Inc.
25 North Avenue
Burlington, Massachusetts 01803
(617) 273-1550

Copyright 1982, Computer Devices, Inc.
All rights reserved.

SYSTEM REQUIREMENTS

Your implementation of Microsoft BASIC requires:

- 48K bytes of memory, minimum:
  32K for Microsoft BASIC
  16K for a Data Segment (BASIC will make use of up to 64 K for a Data Segment, if available)

- 1 disk drive

If your system does not meet these minimum requirements, ask your computer dealer how to expand your system.

Microsoft BASIC is a trademark of Microsoft, Inc.
MS-DOS is a trademark of Microsoft, Inc.
EDLIN, DEBUG, and FILCOM are trademarks of Microsoft, Inc.

INTRODUCTION

BASIC Release 5 from Microsoft is the most extensive implementation of BASIC available for microprocessors. it meets the requirements for the ANSI subset standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, Microsoft BASIC has sophisticated string handling and structured programming features that are especially suited for applications development. And, the Microsoft BASIC language is compatible with the Microsoft BASIC Compiler. Microsoft BASIC gives users what they want from a BASIC -- ease of use plus the features that make a micro perform like a minicomputer or a large mainframe.

In 1975, Microsoft wrote the first BASIC interpreter for the microcomputer. Today Microsoft BASIC, with over 750,000 installations in over 20 operating environments, is recognized as the industry standard. It's the BASIC you'll find on all the largest-selling microcomputers. Many users, manufacturers, and software houses have written applications programs in Microsoft BASIC.

MAJOR FEATURES

1. Four variable types: Integer (±32767), String (up to 255 characters), Single Precision Floating Point (7 digits), Double Precision Floating Point (16 digits)

2. Trace facilities (TRON/TROFF) for easier debugging

3. Error trapping using the ON ERROR GOTO statement

4. PEEK and POKE statements to read and write any memory location

5 Automatic line number generation and renumbering, including referenced line numbers

6. Arrays with up to 8 dimensions

7. Boolean operators OR, AND, NOT, XOR, EQV, and IMP

8. Formatted output using the complete PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion

9. Direct access to the 64K I/O ports with the INP and OUT functions

10. Extensive program editing facilities via EDIT command and EDIT mode subcommands

11. Support of assembly language subroutine calls (up to 10 per program)

12. IF/THEN/ELSE and nested IF/THEN/ELSE constructs

13. Disk BASIC supports variable length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE

NOTE

Features vary from one implementation of Microsoft BASIC to the next. See Chapter 1 of this manual for an exact description of the features in your implementation of Microsoft BASIC.

USING THESE MANUALS

The information in the documents you recieved in this package is divided into reference information and user information.

The Microsoft BASIC Reference Manual contains descriptions of all the features of Microsoft BASIC. The Reference Manual contains no information that is either implementation-specific (that is, applies to a particular microprocessor), or operating system-specific.

The Microsoft BASIC User's Guide contains all of the implementation-specific and operating system-specific information. This information includes telling you which features of Microsoft BASIC are included and which are excluded from your implementation, telling you how your implementation changes the format and the use of some features, and telling you how your operating system affects some features and operation of Microsoft BASIC.

The Microsoft GW-Basic Addendum includes information on language feature extensions to Standard BASIC-86, Version 5.0, for the iAPX 8088 and 8086 microprocessors. GW-BASIC takes advantage of the power of the 16-bit microprocessors by augmenting the standard Microsoft BASIC with three upward levels.

Use the Reference Manual for details of the features. Use the User's Guide to see if a feature differs from its description in the Reference Manual. Also, use the User's Guide for hints about disk I/O and calling assembly language subroutines into your BASIC programs.

SYNTAX NOTATION

The following notation is used throughout these manuals in describing command and statement syntax:

[ ] Square brackets indicate that the enclosed entry is optional.

< > Angle brackets indicate user-entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.

{ } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

... Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

MS-BASIC LANGUAGE MANUAL

TABLE OF CONTENTS

PART I: MS-BASIC USER'S MANUAL

PAGE

CHAPTER 1: FEATURES INCLUDED IN THIS IMPLEMENTATION. .1-1

CHAPTER 2: LANGUAGE DIFFERENCES FOR THIS IMPLEMENTATION. 2-1

2.1 THE DEF SEG STATEMENT. 2-1
2.2 THE CALL STATEMENT. 2-3
2.3 THE INP FUNCTION 2-4
2.4 THE OUT STATEMENT 2-4
2.5 USR FUNCTION CALLS. 2-5

CHAPTER 3: CONVERTING PROGRAMS TO MICROSOFT BASIC . .3-1

3.1 STRING DIMENSIONS 3-1
3.2 MULTIPLE ASSIGNMENTS 3-2
3.3 MULTIPLE STATEMENTS 3-2
3.4 MAT FUNCTIONS 3-2

CHAPTER 4: MICROSOFT BASIC DISK I/O 4-1

4.1 PROGRAM FILE COMMANDS. 4-1
4.2 PROTECTED FILE 4-2
4.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O . .4-2
4.3.1 Sequential Files. 4-3
4.3.1.1 Adding Data to a Sequential File. .4-5
4.3.2 Random Files 4-6
4.3.2.1 Creating a Random File 4-7
4.3.2.2 Assessing a Random File. 4-8

CHAPTER 5: BASIC ASSEMBLY LANGUAGE SUBROUTINES . . .5-1

5.1 MEMORY ALLOCATION 5-1
5.2 THE CALL STATEMENT. 5-2
5.3 USER FUNCTION CALLS 5-7

CHAPTER 6: MICROSOFT BASIC WITH THE MS-DOS OPERATING SYSTEM 6-1

6.1 INITIALIZATION 6-1
6.2 DISK FILES 6-2
6.3 FILES COMMAND 6-3
6.4 RESET COMMAND 6-3
6.5 LOF FUNCTION 6-3
6.6 EOF. 6-4
6.7 MISCELLANEOUS 6-4

INDEX Index-1

PART II: MS-BASIC REFERENCE MANUAL

CHAPTER 1: GENERAL INFORMATION ABOUT MICROSOFT BASIC .1-1

1.1 INITIALIZATION 1-1
1.2 MODES OF OPERATION. 1-1
1.3 LINE FORMAT 1-1
1.3.1 Line Numbers 1-2
1.4 CHARACTER SET 1-3
1.4.1 Control Characters 1-4
1.5 CONSTANTS. 1-4
1.5.1 Single and Double Precision Form for Numeric Constants 1-5
1.6 VARIABLES. 1-6
1.6.1 Variable Names and Declaration Characters .1-6
1.6.2 Array Variables 1-7
1.6.3 Space Requirements 1-8
1.7 TYPE CONVERSION. 1-8
1.8 EXPRESSIONS AND OPERATORS 1-9
1.8.1 Arithmetic Operators 1-10
1.8.1.1 Integer Division and Modulus Arithmetic 1-11
1.8.1.2 Overflow and Division by Zero. . . 1-11
1.8.2 Relational Operators 1-11
1.8.3 Logical Operators 1-12
1.8.4 Functional Operators 1-14
1.8.5 String Operators. 1-15
1.9 INPUT EDITING 1-15
1.10 ERROR MESSAGES 1-16

CHAPTER 2: MICROSOFT BASIC COMMANDS AND STATEMENTS . .2-1

2.1 AUTO 2-2
2.2 CALL 2-3
2.3 CHAIN 2-4
2.4 CLEAR 2-6
2.5 CLOAD 2-7
2.6 CLOSE 2-8
2.7 COMMON. 2-9
2.8 CONT 2-11
2.9 CSAVE 2-12
2.10 DATA 2-13
2.11 DEF FN. 2-14
2.12 DEFINT/SNG/DBL/STR 2-16
2.13 DEF USR 2-17
2.14 DELETE. 2-18
2.15 DIM. 2-19
2.16 EDIT 2-20
2.17 END. 2-24
2.18 ERASE 2-25
2.19 ERR and ERL VARIABLES. 2-26
2.20 ERROR 2-27
2.21 FIELD 2-29
2.22 FOR...NEXT 2-30

2.23 GET. 2-32
2.24 GOSUB...RETURN 2-33
2.25 GOTO 2-34
2.26 IF...THEN [...ELSE] and IF...GOTO. 2-35
2.27 INPUT 2-37
2.28 INPUT#. 2-39
2.29 KILL 2-40
2.30 LET. 2-41
2.31 LINE INPUT 2-42
2.32 LINE INPUT# 2-43
2.33 LIST 2-44
2.34 LLIST 2-46
2.35 LOAD 2-47
2.36 LPRINT and LPRINT USING 2-48
2.37 LSET and RSET 2-49
2.38 MERGE 2-50
2.39 MID$ 2-51
2.40 NAME 2-52
2.41 NEW. 2-53
2.42 NULL 2-54
2.43 ON ERROR GOTO 2-55
2.44 ON...GOSUB and ON...GOTO. 2-56
2.45 OPEN 2-57
2.46 OPTION BASE 2-58
2.47 OUT. 2-59
2.48 POKE 2-60
2.49 PRINT 2-61
2.50 PRINT USING 2-63
2.51 PRINT# and PRINT# USING 2-67
2.52 PUT. 2-69
2.53 RANDOMIZE. 2-70
2.54 READ 2-71
2.55 REM. 2-73
2.56 RENUM 2-74
2.57 RESTORE 2-75
2.58 RESUME. 2-76
2.59 RUN. 2-77
2.60 SAVE 2-78
2.61 STOP 2-79
2.62 SWAP 2-80
2.63 TRON/TROF. 2-81
2.64 WAIT 2-82
2.65 WHILE...WEND. 2-83
2.66 WIDTH 2-84
2.67 WRITE 2-85
2.68 WRITE#. 2-86

CHAPTER 3: MICROSOFT BASIC FUNCTIONS. 3-1

3.1 ABS. 3-2
3.2 ASC. 3-2
3.3 ATN. 3-3
3.4 CDBL 3-3
3.5 CHR$ 3-4
3.6 CINT 3-4

3.7 COS. 3-5
3.8 CSNG 3-5
3.9 CVI, CVS, CVD 3-6
3.10 EOF. 3-6
3.11 EXP. 3-7
3.12 FIX. 3-7
3.13 FRE. 3-8
3.14 HEX$ 3-8
3.15 INKEY$. 3-9
3.16 INP. 3-9
3.17 INPUT$. 3-10
3.18 INSTR 3-11
3.19 INT. 3-12
3.20 LEFT$ 3-12
3.21 LEN. 3-13
3.22 LOC. 3-13
3.23 LOG. 3-14
3.24 LPOS 3-14
3.25 MID$ 3-15
3.26 MKI$, MKS$, MKD$ 3-15
3.27 OCT$ 3-16
3.28 PEEK 3-16
3.29 POS. 3-17
3.30 RIGHT$. 3-17
3.31 RND. 3-18
3.32 SGN. 3-18
3.33 SIN. 3-19
3.34 SPACE$. 3-19
3.35 SPC. 3-20
3.36 SQR. 3-20
3.37 STR$ 3-21
3.38 STRING$ 3-21
3.39 TAB. 3-22
3.40 TAN. 3-22
3.41 USR. 3-23
3.42 VAL. 3-23
3.43 VARPTR. 3-24

APPENDICES

APPENDIX A: SUMMARY OF ERROR CODES AND ERROR MESSAGES A-1

APPENDIX B: MATHEMATICAL FUNCTIONS B-1

APPENDIX C: ASCII CHARACTER CODES. C-1

INDEX Index-1

PART 3: GW-BASIC ADDENDUM

LANGUAGE FEATURE EXTENSIONS TO STANDARD
BASIC-86, VERSION 5
FOR THE
iAPX 8088/8086 MICROPROCESSORS

CHAPTER 1: GW-BASIC LANGUAGE FEATURES FOR THE iAPX 88/86 1-1

1.1 THE BASIC FULL SCREEN EDITOR 1-2
1.1.1 Inputting the BASIC Program 1-2
1.1.2 Altering Lines with the Editor 1-3
1.1.3 Changing a BASIC Program 1-8
1.1.4 The EDIT Statement 1-10
1.1.5 The New Error Messages. 1-11
1.2 LEVEL 1 GW-BASIC STATEMENT EXTENSIONS . . . 1-12
1.2.1 The BEEP Statement 1-12
1.2.2 The SOUND Statement. 1-13
1.2.3 The BLOAD and BSAVE Statements 1-14
1.2.3.1 The BLOAD Statement 1-14
1.2.3.2 The BSAVE Statement 1-17
1.2.4 The CALL Statement 1-19
1.2.5 THE COLOR Statement. 1-22
1.2.5.1 The COLOR Statement in Alpha Modes . 1-22
1.2.6 The CLS Statement 1-25
1.2.7 The CSRLIN Function. 1-26
1.2.8 The DEF SEG Statement 1-27
1.2.9 The KEY Statement 1-29
1.2.10 The Command Super Shift Keys. 1-31
1.2.11 The LOCATE Statement 1-32
1.2.12 The LIST Statement 1-34
1.2.13 The GRAPHIC Statement 1-36
1.2.13.1 The PSET Statement 1-39
1.2.13.2 The PRESET Statement. 1-38
1.2.13.3 The POINT Function 1-40
1.2.13.4 The LINE Statement 1-41
1.2.14 The SCREEN Statement 1-43
1.2.14.1 The SCREEN Function 1-45
1.2.15 The WIDTH Statement. 1-46
1.3 LEVEL 2 / GW-BASIC INITIALIZATION. 1-48
1.3.1 Level 2 Initialization. 1-48
1.3.2 Error Messages 1-51
1.3.3 GW-BASIC Statement Extensions 1-53
1.3.3.1 The DATE$ Statement 1-53
1.3.3.2 The TIME$ Statement 1-55
1.3.3.3 The OPEN Statement 1-57
1.3.4 Disk Files. 1-60
1.3.4.1 The LOF Function 1-60
1.3.4.2 The VARPTR Function 1-61
1.3.5 The RS232 Communication Option 1-63
1.3.5.1 OPENing a COM File 1-63

1.3.5.2 Communication I/O. 1-65
1.3.5.2.1 The COM I/O Functions. . . . 1-65
1.3.5.2.2 The INPUT$ Function 1-66
1.3.5.2.3 The GET and PUT Statements . . 1-67
1.3.6 Notes on the TTY Programming Example . . 1-69
1.4 GW-BASIC / LEVEL 3. 1-73
1.4.1 Level 3 Overview. 1-73
1.4.2 Event Trapping 1-74
1.4.3 Advanced Graphics 1-77
1.4.4 PLAY <string expression> 1-84

MS-BASIC USER'S GUIDE

for 8086 and 8088 microprocessors and the MS-DOS operating system

CHAPTER 1

FEATURES INCLUDED IN THIS IMPLEMENTATION

All statements, commands, and functions described in the Microsoft BASIC Reference Manual are implemented unless listed below. Some of the statements, commands, or functions may be affected by this particular implementation. See Chapter 2, Language Differences for This Implementation.

The following features are not included in the 8086 Version of Microsoft BASIC:

CLOAD CSAVE

The following statement is included in the 8086 version of Microsoft BASIC:

DEF SEG

DEF SEG and changes to some other statements and functions are described in Chapter 2, below.

CHAPTER 2

LANGUAGE DIFFERENCES FOR THIS IMPLEMENTATION

The following features differ in this implementation from the descriptions given in the Microsoft BASIC Reference Manual.

2.1 THE DEF SEG STATEMENT

The DEF SEG statement is not included in the Microsoft BASIC Reference Manual.

Format: DEF SEG [=<address>]

where: address is a valid numeric expression returning an unsigned integer in the range 0 to 65535.

Purpose: To assign the current segment address to be referenced by a subsequent CALL (see Section 2.2), a USR function call, or a PEEK or POKE statement.

Remarks: The address specified is saved for use as the segment required by PEEK, POKE, and CALL statements.

Entry of any value outside the <address> range 0-65535 will result in an "Illegal Function Call" error, and the previous value will be retained.

If the <address> option is omitted, the segment to be used is set to BASIC's data segment (DS). This is the initial default value.

If the <address> option is given, it should be based on a 16-byte boundary. For PEEK, POKE, or CALL statements, the value is shifted left 4 bits (this is done by the microprocessor, not by BASIC) to form the code segment address for the

subsequent call instruction. BASIC does not perform additional checking to assure that the resultant segment address is valid.

NOTE: DEF and SEG MUST be separated by a space. Otherwise, BASIC would interpret the statement DEFSEG=100 to mean, "assign the value 100 to the variable DEFSEG."

Example:

```
10 DEF SEG=&HB800 'Set segment to Screen buffer
20 DEF SEG 'Restore segment to BASIC's DS
```

2.2 THE CALL STATEMENT

Format: CALL <variable name> [(<argument list>)]

where: variable name contains the segment offset that is the starting point in memory of the subroutine being CALLed. Note that the variable name must be assigned to the segment offset before the CALL statement is issued (see example below).

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is the recommended way of calling 8086 machine language programs with Microsoft BASIC. It is suggested that the old style user-call USR(n) not be used. See Chapter 5 for comparison of the two methods and for a complete description of using the CALL statement for assembly language subroutines.

When a CALL statement is executed, control is transferred to the user's routine via the segment address given in the last DEF SEG statement and the segment offset specified by the <variable name> portion of the CALL statement. Values are returned to BASIC by including the variable name which will receive the result in the <argument list>.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. Microsoft BASIC follows the rules described for the MEDIUM case.

Example:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment address to 8000 Hex. The variable FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

2.3 THE INP FUNCTION

Format: INP(I)

where: I is a valid machine port number in the range 0 to 65535.

Purpose: To return the byte read from port I.

Remarks: INP is the complementary function to the OUT statement (Section 2.4 below).

Example: 100 A=INP(54321)

In assembly language, this is equivalent to:

```
MOV  DX,54321
IN   AL,DX
```

2.4 THE OUT STATEMENT

Format: OUT I,J

where: I and J are integer expressions in the range 0 to 65535. I is a machine port number, and J is the data to be transmitted.

Purpose: To send a byte to a machine output port.

Remarks: OUT is the complementary statement to the INP function (Section 2.3 above).

Example: 100 OUT 12345,225

In assembly language, this is equivalent to:

```
MOV  DX,12345
MOV  AL,255
OUT  DX,AL
```

2.5 USR FUNCTION CALLS

Although the CALL statement is recommended for calling assembly language subroutines, the USR function call may also be used. See Chapter 5 for comparison of CALL and USR and for a detailed discussion of calling assembly language subroutines.

Format: USR[<digit>][(argument)]

where: digit specifies which USR routine is being called. (See the Microsoft BASIC Reference Manual, Section 2.13, DEF USR, for the rules governing <digit>.) If <digit> is omitted, USR0 is assumed.

argument is any numeric or string expression.

Purpose: To call an assembly language subroutine.

Remarks: In this implementation, if a segment other than the default segment (BASIC's data segment, DS) is to be used, a DEF SEG statement MUST be executed prior to a USR call. This assures that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (See the DEF SEG statement in Section 2.1 above.)

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG segment address determine the starting address of the subroutine.

Example:

```
100 DEF SEG=&H8000
110 DEF USR0=0
120 X=5 'Note that X is single precision
130 Y = USR0(X)
140 PRINT Y
```

Note that the type (numeric or string) of the variable receiving the function call must be consistent with the argument passed; or it may be set to Integer by calling MAKINT in the user's routine before returning to BASIC.

CHAPTER 3

CONVERTING PROGRAMS TO MICROSOFT BASIC

If you have programs written in a BASIC other than Microsoft BASIC, some minor adjustments may be necessary before running them with Microsoft BASIC. Here are some specific things to look for when converting BASIC programs.

3.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Microsoft BASIC statement DIM A$(J).

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Microsoft BASIC string concatenation.

In Microsoft BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Microsoft BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	Microsoft BASIC
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

3.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. Microsoft BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

3.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With Microsoft BASIC, be sure all statements on a line are separated by a colon (:).

3.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

CHAPTER 4

MICROSOFT BASIC DISK I/O

Disk I/O procedures for the beginning BASIC user are examined in this chapter. If you are new to BASIC or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The MS-DOS operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

4.1 PROGRAM FILE COMMANDS

The following list reviews the commands and statements used in program file manipulation.

SAVE <filename>[,A] — Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

(See Section 4.2 below for discussion of how to SAVE protected files.)

LOAD <filename>[,R] — Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADing. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files. (LOAD <filename>,R and RUN <filename>,R are equivalent.)

RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open. (RUN <filename>,R and LOAD <filename>,R are equivalent.)
MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename> AS<new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

4.2 PROTECTED FILE

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

4.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC program: sequential files and random access files.

4.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is a series of ASCII characters stored, one item after another (sequentially), in the order it is sent. It is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode.	`OPEN "O",#1,"DATA"`
2. Write data to the file using the PRINT# statement. (WRITE# may be used instead.)	`PRINT#1,A$;B$;C$`
3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode.	`CLOSE #1` `OPEN "I",#1,"DATA"`
4. Use the INPUT# statement to read data from the sequential file into the program.	`INPUT#1,X$,Y$,Z$`

Program 1 is a short program that creates a sequential file, "DATA", from information you input at the terminal:

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM 1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program 2. It accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978:

```
10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM 2 - ACCESSING A SEQUENTIAL FILE

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed.

4.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES":

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program 3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM 3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

4.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI$	CVI		
MKS$	CVS		
MKD$	CVD		

4.3.2.1 Creating A Random File -

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD #1, 20 AS N$,
 4 AS A$, 8 AS P$
```

3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single precision value, and MKD$ for a double precision value.

```
LSET N$=X$
LSET A$=MKS$(AMT)
LSET P$=TEL$
```

4. Write the data from the buffer to the disk using the PUT statement.

```
PUT #1,CODE%
```

Look at Program 4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM 4 - CREATE A RANDOM FILE

4.3.2.2 Accessing A Random File -

The following program steps are required to access a random file:

1. OPEN the file in "R" mode.

```
OPEN "R",#1,"FILE",32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

```
FIELD #1 20 AS N$,
       4 AS A$, 8 AS P$
```

NOTE

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

```
GET #1,CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values.

```
PRINT N$
PRINT CVS(A$)
```

Program 5 accesses the random file "FILE" that was created in Program 4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed:

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM 5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file#1 is higher than 50.

Program 6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
125 FIELD#1,1 AS F$,30 AS D$, 2 AS Q$,2 AS R$,4 AS P$
130 PRINT:PRINT "FUNCTIONS:":PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
225 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT
        "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT#1,PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$###.##";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;" IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
        " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
         CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
        GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM 6 - INVENTORY

CHAPTER 5

BASIC ASSEMBLY LANGUAGE SUBROUTINES

All versions of Microsoft BASIC have provisions for interfacing with assembly language subroutines via the USR function and the CALL statement.

The USR function allows assembly language subroutines to be called in the same way BASIC Intrinsic functions are called. However, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is compatible with more languages than is the USR function call, it produces more readable source code, and it can pass multiple arguments.

5.1 MEMORY ALLOCATION

IMPORTANT

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s) with the /M: switch.

In addition to the BASIC interpreter code area, Microsoft BASIC uses up to 64K of memory beginning at its data (DS) segment.

If, when an assembly language subroutine is called, more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the operating system or the BASIC POKE statement. If the user has the Microsoft Utility Software Package, the routines may be assembled with the MACRO-86

assembler and linked using the MS-LINK Linker, but not loaded. To load the program file, the user should observe these guidelines:

The subroutines must not contain any long references.

Skip over the first 512 bytes of the MS-LINK output file, then read in the rest of the file.

5.2 THE CALL STATEMENT

As mentioned earlier, the CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is further suggested that the old style user-call USR(n) not be used.

Format:

CALL <variable name> [(<argument list>)]

where: variable name contains the segment offset that is the starting point in memory of the subroutine being CALLed.

argument list contains the variables or constants, separated by commas, that are to be passed to the routine.

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler Operator's Manual. BASIC follows the rules described for the MEDIUM case (summarized in the following discussion).

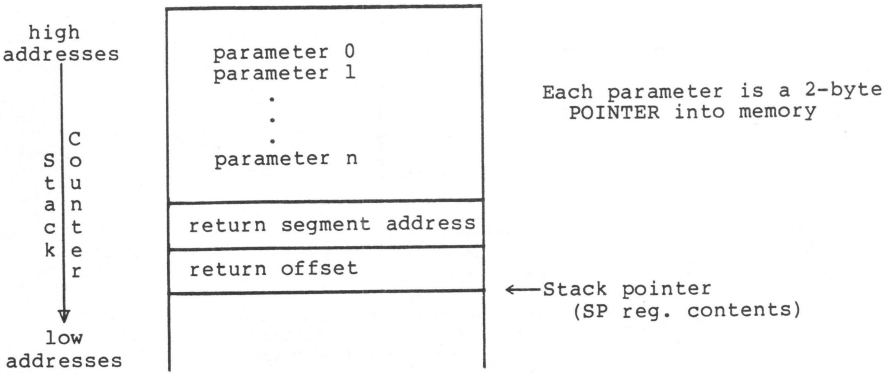
Invoking the CALL statement causes the following to occur:

1. For each parameter in the argument list, the 2 byte offset of the parameter's location within the data segment (DS) is pushed onto the stack.

2. BASIC's return address code segment (CS), and offset (IP) are pushed onto the Stack.

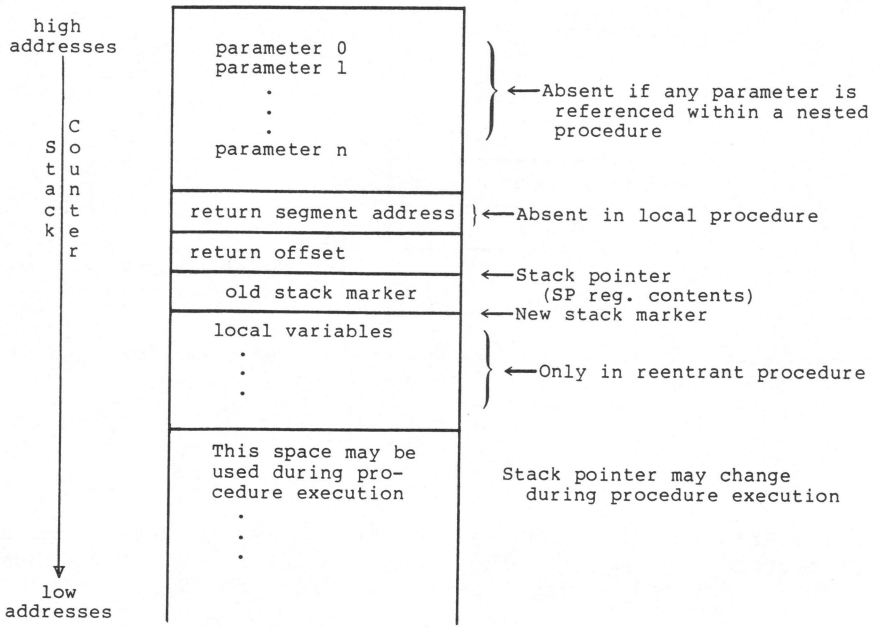
3. Control is transferred to the user's routine via an 8086 long call to the segment address given in the last DEF SEG statement and the offset given in <variable name>.

These actions are illustrated by the two following diagrams, which illustrate first, the state of the stack at the time of the CALL statement, and second, the condition of the stack during execution of the CALLed subroutine.



Stack Layout when CALL statement is activated

The user's routine now has control. Parameters may be referenced by moving the Stack pointer (SP) to the Base Pointer (BP) and adding a positive offset to (BP).



Stack Layout during Execution of of a CALL statement

You must observe the following rules when coding a subroutine:

1. The CALLed routine may destroy the AX, BX, CX, DX, SI, DI, and BP registers.

2. The CALLed program MUST know the number and length of the parameters passed. References to parameters are positive offsets added to (BP) (assuming the called routine moved the current stack pointer into BP; i.e., MOV BP,SP). That is, the location of p1 is at 8(BP), p2 is at 6(BP), p3 is at 4(BP), ...etc.

3. The CALLed routine must do a RET <n> (where <n> is two times the number of parameters in the argument list) to adjust the stack to the start of the calling sequence.

4. Values are returned to BASIC by including in the argument list the variable name which will receive the result.

5. If the argument is a string, the parameter's offset points to 3 bytes called the "String Descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

IMPORTANT

If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

```
20 A$ = "BASIC"+""
```

This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

6. Strings may be altered by user routines, but the length MUST NOT be changed. BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

Example:

```
100 DEF SEG=&H8000
110 FOO=&H7FA
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment to 8000 Hex. The value of variable FOO is added into the address as the low word after the DEF SEG value is left shifted 8 bits. (This is a function of the microprocessor, not of BASIC.) Here, FOO is set to &H7FA, so that the call to FOO will execute the subroutine at location 8000:7FA Hex (absolute address 807FA Hex).

The following sequence of 8086 assembly language demonstrates access of the parameters passed and storing a return result in the variable 'C'.

```
MOV     BP,SP       ;Get current Stack posn in BP.
MOV     BX,6[BP]    ;Get address of B$ dope.
MOV     CL,[BX]     ;Get length of B$ in CL.
MOV     DX,1[BX]    ;Get addr of B$ text in DX.
        .
        .
        .
MOV     SI,8[BP]    ;Get address of 'A' in SI.
MOV     DI,4[BP]    ;Get pointer to 'C' in DI.
MOVS    WORD        ;Store variable 'A' in 'C'.
RET     6           ;Restore Stack, return.
```

IMPORTANT

> The called program must know the variable type for numeric parameters passed. In the above example, the instruction
>
> MOVS WORD
>
> will copy only 2 bytes. This is fine if variables A and C are integer. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

5.3 USR FUNCTION CALLS

Although the CALL statement is the recommended way of calling assembly language subroutines, the USR function call is still available for compatibility with previously-written programs.

The format of the USR function call is:

```
USR[<digit>][(argument)]
```

where: digit is from 0 to 9. <digit> specifies which USR routine is being called (see DEF USR statement, Section 2.13 of the Reference Manual). If <digit> is omitted, USR0 is assumed.

argument is any numeric or string expression. Arguments are discussed in detail below.

In this implementation of BASIC, a DEF SEG statement MUST be executed prior to a USR call to assure that the code segment points to the subroutine being called. The segment address given in the DEF SEG statement determines the starting segment of the subroutine. (See the DEF SEG statement in Chapter 2.)

For each USR function, a corresponding DEF USR statement must have been executed to define the USR call offset. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register [AL] contains a value which specifies the type of argument that was given. The value in [AL] may be one of the following:

2 Two-byte integer (two's complement)

3 String

4 Single precision floating point number

8 Double precision floating point number

If the argument is a number, the [BX] register pair points to the Floating Point Accumulator (FAC) where the argument is stored:

FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative).

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument.

FAC-3 contains the lower 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-2 contains the middle 8 bits of mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating point number:

FAC-7 - FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string:

the [DX] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in BASIC's Data Segment.

IMPORTANT

> If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. See the CALL statement above.
>
> Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

Example:

```
110 DEF USR0=&H8000 'Assumes user gave /M:32767
120 X=5 'Note that X is single precision
130 Y = USR0(X)
140 PRINT Y
```

Note that the type (numeric or string) of the variable receiving the function call must be consistent with the argument passed; or it may be set to Integer by calling MAKINT in the user's routine before returning to BASIC.

We have loaded the following assembly language routine to simply multiply the argument passed by 2 and return an integer result.

Always be sure that your programs are defined by a PROC FAR statement.

```
DOUBLE          SEGMENT

                ASSUME  CS:DOUBLE

FRCINTOFFSET    EQU     103H
MAKINTOFFSET    EQU     107H

FRCINT          LABEL   DWORD
                DW      FRCINTOFFSET
FRCSEG          DW      ?

MAKINT          LABEL   DWORD
                DW      MAKINTOFFSET
MAKSEG          DW      ?

USRPRG          PROC    FAR
                POP     SI
                POP     AX          ;Recover BASIC's CS
                PUSH    AX
                PUSH    SI
                MOV     FRCSEG,AX   ;Set segment for long indirect CALL
                MOV     MAKSEG,AX
                CALL    FRCINT      ;Force arg in FAC to int in [BX]
                ADD     BX,BX       ; [BX] = [BX] * 2
                CALL    MAKINT      ;Put Result back in FAC
                RET                 ;Long return to BASIC
USRPRG          ENDP

DOUBLE          ENDS
```

When FRCINT or MAKINT is called and when the subroutine terminates with a return, ES, DS, and SS must have the same

value they had when the subroutine was entered. These registers point to BASIC's Data Segment.

CHAPTER 6

MICROSOFT BASIC WITH THE MS-DOS OPERATING SYSTEM

The MS-DOS version of Microsoft BASIC is supplied on a standard size 3740 single density diskette. The name of the file is BASIC86.COM. (A 48K or larger MS-DOS system is recommended.)

To run BASIC, bring up MS-DOS and type the following:

```
A>BASIC86 <carriage return>
```

The system will reply:

```
xxxx Bytes Free
Microsoft BASIC Version 5.x
[MS-DOS Version]
Copyright 1977-1981 (C) by Microsoft, Inc.
Created: dd-mmm-yy
Ok
```

6.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the BASIC86 command to MS-DOS. The format of the command line is (the command line may appear differently on your screen than on this page):

```
A>BASIC86 [<filename>][/F:<number of files>]
      [/M:<highest memory location>][/S:<maximum record size>]
```

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the BATCH facility of MS-DOS. Such programs should include a SYSTEM statement (see below) to return to MS-DOS when they have finished, allowing the next program in the batch stream to execute.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by BASIC.

/S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE

<number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

```
A>BASIC86 PAYROLL.BAS        Use all memory and 3 files,
                             load and execute PAYROLL.BAS.

A>BASIC86 INVENT/F:6         Use all memory and 6 files,
                             load and execute INVENT.BAS.

A>BASIC86 /M:32768           Use first 32K of memory and
                             3 files.

A>BASIC86 DATACK/F:2/M:&H9000
                             Use first 36K of memory, 2
                             files, and execute DATACK.BAS.
```

6.2 DISK FILES

Disk filenames follow the normal MS-DOS naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive; otherwise the currently selected drive is assumed. A default extension of .BAS is used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

Large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

6.3 FILES COMMAND

Format: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:
```
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"
```

6.4 RESET COMMAND

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

6.5 LOF FUNCTION

Format: LOF(<file number>)

Purpose: Returns the length of the file in bytes.

Example:
```
110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"
```

6.6 EOF

Format: EOF (<file number>)

Purpose: Returns -1 (true) if the end of a sequential or random file has been reached. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
```

6.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.

2. To return to MS-DOS, use the SYSTEM command or statement. SYSTEM closes all files and then returns to MS-DOS command. Control-C always returns to BASIC, not to MS-DOS.

3. FRCINT is at 103 hex and MAKINT is at 107 hex.

INDEX

Assembly Language Subroutines 5-1

BAS 4-1
BASIC86.COM 6-1

CALL 2-3, 5-2
CLOSE 4-3, 4-7
CVD 4-7
CVI 4-7
CVS 4-7

DEF SEG 2-1, 5-2, 5-7
DEF USR 5-7

EOF 4-3, 4-5, 6-4
Error trapping 4-6

FIELD 4-7
FILES 6-3
Floating Point Accumulator . . 5-7
FRCINT 6-4

GET 4-7, 6-4

INP 2-4
INPUT 4-8
INPUT# 4-3

KILL 4-2

Language Differences
from Reference Manual . . . 2-1
Language Differences 2-1
LET 4-8
LINE INPUT# 4-3
LOAD 4-1
LOC 4-3, 4-5, 4-7
LOF 6-3
LSET 4-7

MACRO-86 5-2
MAKINT 6-4
Memory Allocation 5-1
MERGE 4-2
MID$ 3-1
MKD$ 4-7
MKI$ 4-7
MKS$ 4-7
MS-DOS 4-1, 6-1
MS-LINK 5-2

OPEN 4-3, 4-7
OUT 2-4

PRINT# USING 4-5
PRINT# USING 4-3
PRINT# 4-3
Protected files 4-2
PUT 4-7

Random files 4-6, 6-4
RESET 6-3
RET 5-5
RSET 4-7
RUN 4-2

SAVE 4-1
Sequential files 4-3
String functions 3-1
String literal 5-5, 5-8
String space 4-8
Syntax Notation 6
SYSTEM 6-4

USR Function Calls 5-7

WRITE# 4-3

Index of Files on Diskette

Your Microsoft BASIC diskette contains these files:

BASIC86.COM - contains the BASIC interpreter
RANDOM.BAS - tests BASIC's random I/O logic

MS-BASIC REFERENCE MANUAL

CHAPTER 1

GENERAL INFORMATION ABOUT MICROSOFT BASIC

1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of Microsoft BASIC. Check the Microsoft BASIC User's Guide for your machine to determine how Microsoft BASIC is initialized with your operating system.

1.2 MODES OF OPERATION

When Microsoft BASIC is initialized, it types the prompt "Ok". "Ok" means BASIC is at command level; that is, it is ready to accept commands. At this point, Microsoft BASIC may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement[:BASIC statement...] <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A Microsoft BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 255 characters.

It is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>.

1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The Microsoft BASIC character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in Microsoft BASIC are the upper case and lower case letters of the alphabet.

The numeric characters in Microsoft BASIC are the digits 0 through 9.

The following special characters and terminal keys are recognized by Microsoft BASIC:

Character	Name
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
_	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

The following control characters are in Microsoft BASIC:

Control-A	Enters Edit Mode on the line being typed.
Control-C	Interrupts program execution and returns to BASIC command level.
Control-G	Rings the bell at the terminal.
Control-H	Backspace. Deletes the last character typed.
Control-I	Tab. Tab stops are every eight columns.
Control-O	Halts program output while execution continues. A second Control-O restarts output.
Control-R	Retypes the line that is currently being typed.
Control-S	Suspends program execution.
Control-Q	Resumes program execution after a Control-S.
Control-U	Deletes the line that is currently being typed.

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1.	Integer constants	Whole numbers between -32768 and +32767. Integer constants do not have decimal points.
2.	Fixed Point constants	Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38.
Examples:

235.988E-7 = .0000235988
2359E6 = 2359000000

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

&H76
&H32F

5. Octal constants

Octal numbers with the prefix &O or &. Examples:

&O347
&1234

1.5.1 Single And Double Precision Form For Numeric Constants

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 6 digits. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

Single Precision Constants	Double Precision Constants
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names And Declaration Characters

Microsoft BASIC variable names may be any length; up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word, but embedded reserved words are allowed. Embedded reserved words are allowed. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all Microsoft BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string.

String variable names are written with a dollar sign ($) as the last character. For example: A$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single or double precision values. The type declaration characters for these variable names are as follows:

%	Integer variable
!	Single precision variable
#	Double precision variable

The default type for a numeric variable name is single precision.

Examples of Microsoft BASIC variable names follow.

PI#	declares a double precision value
MINIMUM!	declares a single precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single precision value

There is a second method by which variable types may be declared. The BASIC statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

1.6.3 Space Requirements

VARIABLES:	BYTES
INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8

ARRAYS:	BYTES
INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)
   Example:

```
10 A% = 23.42
20 PRINT A%
RUN
 23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.
   Examples:

```
10 D# = 6#/7      The arithmetic was performed
20 PRINT D#       in double precision and the
RUN               result was returned in D#
 .8571428571428571 as a double precision value.
```

```
10 D = 6#/7
20 PRINT D
RUN
 .857143
```

The arithmetic was performed in double precision and the result was returned to D (single precision variable), rounded and printed as a single precision value.

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.

4. When a floating point value is converted to an integer, the fractional portion is rounded. Example:

```
10 C% = 55.88
20 PRINT C%
RUN
 56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value. Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04   2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories:

1. Arithmetic

2. Relational
3. Logical
4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*,/	Multiplication, Floating Point Division	X*Y X/Y
+,-	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	BASIC Expression
X+2Y	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic -
Two additional operators are available in Microsoft BASIC: Integer division and modulus arithmetic.

Integer division is denoted by the baskslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.
For example:

```
10\4 = 2
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero -
If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then used to make a decision regarding program flow. (See IF, Section 2.26.)

Operator	Relation Tested	Expression
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X+Y < (T-1)/Z
```

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

```
NOT
     X     NOT X
     1       0
     0       1

AND
     X     Y     X AND Y
     1     1        1
     1     0        0
     0     1        0
     0     0        0

OR
     X     Y     X OR Y
     1     1        1
     1     0        1
     0     1        1
     0     0        0

XOR
     X     Y     X XOR Y
     1     1        0
     1     0        1
     0     1        1
     0     0        0

EQV
     X     Y     X EQV Y
     1     1        1
     1     0        0
     0     1        0
     0     0        1

IMP
     X     Y     X IMP Y
     1     1        1
     1     0        0
     0     1        1
     0     0        1
```

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is

performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 111111 and 16 = binary 10000, so 63 AND 16 = 16
15 AND 14=14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 = binary 1111111111111111 and 8 = binary 1000, so -1 AND 8 = 8
4 OR 2=6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110)
10 OR 10=10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2=-1	-1 = binary 1111111111111111 and -2 = binary 1111111111111110, so -1 OR -2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Microsoft BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of Microsoft BASIC's intrinsic functions are described in Chapter 3.

Microsoft BASIC also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.11.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

```
=    <>    <    >    <=    >=
```

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided. See EDIT, Section 2.16.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of Microsoft BASIC error codes and error messages, see Appendix A.

CHAPTER 2

MICROSOFT BASIC COMMANDS AND STATEMENTS

All of the Microsoft BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction. See below for format notation.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example:

```
AUTO 100,50        Generates line numbers 100,
                   150, 200 ...

AUTO               Generates line numbers 10,
                   20, 30, 40 ...
```

2.2 CALL

Format: CALL <variable name>[(<argument list>)]

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.40)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:
```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
     .
     .
     .
```

NOTE: For a BASIC Compiler program, line 110 is not needed because the address of MYROUT will be assigned by the linking loader at load time.

2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,[<line number exp>] [,ALL][,DELETE<range>]]

Purpose: To call a program and pass variables to it from the current program.

Remarks: <filename> is the name of the program that is called. Example:

```
CHAIN"PROG1"
```

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. Example:

```
CHAIN"PROG1",1000
```

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7. Example:

```
CHAIN"PROG1",1000,ALL
```

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGEd. Example:

```
CHAIN MERGE"OVRLAY",1000
```

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

```
CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000
```

The line numbers in <range> are affected by the RENUM command.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

NOTE: The Microsoft BASIC compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement format is CHAIN <filename>. If you wish to maintain compatibility with the Microsoft BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during CHAINing.

NOTE: When using the MERGE option, user-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.

2.4 CLEAR

Format: CLEAR [,[<expression1>][,<expression2>]]

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks: <expression1> is a memory location which, if specified, sets the highest location available for use by BASIC.

<expression2> sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: BASIC allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

NOTE: The Microsoft BASIC Compiler supports the CLEAR statement with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Examples:

```
CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000
```

2.5 CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC prints Ok. If not, BASIC prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is possitioned correctly.

See also CSAVE, Section 2.9.

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC.

Example: CLOAD "MAX2"

Loads file "M" into memory.

2.6 CLOSE

Format: CLOSE[[#]<file number>[,[#]<file number...>]]

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

2.7 COMMON

Format: COMMON <list of variables>

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:
```
100 COMMON A,B,C,D(),G$
110 CHAIN "PROG3",10
     .
     .
     .
```

NOTE: The Microsoft BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. The current non-executable statements are:

```
COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
%INCLUDE
```

Arrays in COMMON must be declared in preceding DIM statements,

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN style named COMMON areas are also supported; however, the variables are not preserved across CHAINs. The syntax for named COMMON is as follows:

```
COMMON /<name>/ <list of variables>
```

where <name> is 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to explicityly pass parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed-to

programs. With the Microsoft BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the %INCLUDE statement in each program. For example:

```
MENU.BAS
        10 %INCLUDE COMDEF
        .
        .
        . 1000 CHAIN "PROG1"


PROG1.BAS
        10 %INCLUDE COMDEF
        .
        .
        . 2000 CHAIN "MENU"


COMDEF.BAS
        100 DIM A(100),B$(200)
        110 COMMON I,J,K,A,()
        120 COMMON A$,B$,(),X,Y,Z
```

2.8 CONT

Format: CONT

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break.

Example: See example Section 2.61, STOP.

2.9 CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD, Section 2.5.

NOTE: CSAVE and CLOAD are not included in all implementations of BASIC.

Example: CSAVE "TIMER"

Saves the program currently in memory on cassette under filename "T".

2.10 DATA

Format: DATA <list of constants>

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.54)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example: See examples in Section 2.54, READ.

2.11 DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call.

User-defined functions may be numeric or string. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
        .
        .
410 DEF FNAB(X,Y)=X^3/Y^2
420 T=FNAB(I,J)
        .
        .
```

Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC assumes all variables without declaration characters are single precision variables.

Examples:

```
10 DEFDBL L-P
```
All variables beginning with the letters L, M, N, O, and P will be double precision variables.

```
10 DEFSTR A
```
All variables beginning with the letter A will be string variables.

```
10 DEFINT I-N,W-Z
```
All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.13 DEF USR

Format: DEF USR[<digit>]=<integer expression>

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Assembly Language Subroutines, in the Microsoft BASIC User's Guide.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
.
.
.
200 DEF USR0=24000
210 X=USR0(Y^2/2.89)
.
.
.
```

2.14 DELETE

Format: DELETE[<line number>][-<line number>]

Purpose: To delete program lines.

Remarks: BASIC always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples:

DELETE 40	Deletes line 40
DELETE 40-100	Deletes lines 40 through 100, inclusive
DELETE-40	Deletes all lines up to and including line 40

2.15 DIM

Format: DIM <list of subscripted variables>

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
    .
    .
    .
```

2.16 EDIT

Format: EDIT <line number>

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retyping the entire line. Upon entering Edit Mode, BASIC types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and $ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space — Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

Rubout — In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I — I<text>$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X — The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D — [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H — H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S — The subcommand [i]S<ch> searches for the ith

occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K — The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C — The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> — Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E — The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q — The Q subcommand returns to BASIC command level, without saving any of the changes that were made to the line during Edit Mode.

L — The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A — The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination , first exit Edit Mode with the Q subcommand. BASIC will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.17 END

Format: END

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

2.18 ERASE

Format: ERASE <list of array variables>

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Redimensioned array" error occurs.

NOTE: The Microsoft BASIC compiler does not support ERASE.

Example:

```
.
.
.
450 ERASE A,B
460 DIM B(99)
.
.
.
```

2.19 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ...
Otherwise, use

```
IF ERR = error code THEN ...

IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. The Microsoft BASIC error codes are listed in Appendix A.

2.20 ERROR

Format: ERROR <integer expression>

Purpose: 1) To simulate the occurrence of a BASIC error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC (see Appendix A), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by the Microsoft BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to Microsoft BASIC.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1:

```
LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30
```

Or, in direct mode:

```
Ok
ERROR 15              (you type this line)
String too long       (BASIC types this line)
Ok
```

Example 2:

```
.
.
.
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
.
.
.
```

2.21 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>..

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.22 FOR...NEXT

Format:

```
FOR <variable>=x TO y [STEP z]
 .
 .
 .
NEXT [<variable>][,<variable>...]
```

where x, y and z are numeric expressions.

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT

statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1:

```
10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  20
 3  30
 5  40
 7  50
 9  60
Ok
```

Example 2:

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3:

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of Microsoft BASIC set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

2.23 GET

Format: GET [#]<file number>[,<record number>]

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

2.24 GOSUB...RETURN

Format:

```
GOSUB <line number>
    .
    .
    .
RETURN
```

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertant entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Format: GOTO <line number>

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>

[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>

[ELSE <statement(s)> | <line number>]

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
     THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1:

```
200 IF I THEN GET#1,I
```

This statement GETs record number I if I is not zero.

Example 2:

```
100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"
     .
     .
     .
```

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

2.27 INPUT

Format: INPUT[;][<"prompt string">;]<list of variables>

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the messsage "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
? 5      (The 5 was typed in by the user
          in response to the question mark.)
 5 SQUARED IS 25
Ok


LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS?  7.4  (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Format: INPUT#<file number>,<variable list>

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

2.29 KILL

Format: KILL <filename>

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.30 LET

Format: [LET] <variable>=<expression>

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
     .
     .
     .
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
     .
     .
     .
```

2.31 LINE INPUT

Format: LINE INPUT[;][<"prompt string">;]<string variable>

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into <string variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See Example, Section 2.32, LINE INPUT#.

2.32 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES    234,4   MEMPH
LINDA JONES    234,4    MEMPHIS
Ok
```

2.33 LIST

Format 1: LIST [<line number>]

Format 2: LIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, only the specified line will be listed.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.

2. If only the second number is specified, all lines from the beginning of the program through that line are listed.

3. If both numbers are specified, the entire range is listed.

Examples: Format 1:

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Format 2:

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.34 LLIST

Format: LLIST [<line number>[-[<line number>]]]

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of Microsoft BASIC.

Example: See the examples for LIST, Format 2.

2.35 LOAD

Format: LOAD <filename>[,R]

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example: LOAD "STRTRK",R

2.36 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

LPRINT USING <string exp>;<list of expressions>

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

LPRINT assumes a 132-character-wide printer.

NOTE: LPRINT and LLIST are not included in all implementations of Microsoft BASIC.

2.37 LSET AND RSET

Format: LSET <string variable> = <string expression>
RSET <string variable> = <string expression>

Purpose: To move data from memory to a random file buffer (in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than were FIELDed to <string variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See the MKI$, MKS$, MKD$ functions, Section 3.25.

Examples:
```
150 LSET A$=MKS$(AMT)
160 LSET D$=DESC($)
```

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

NOTE: LSET or RSET may also be used with a non-fielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

2.38 MERGE

Format: MERGE <filename>

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.) The file must have been SAVEd in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC always returns to command level after executing a MERGE command.

Example: MERGE "NUMBRS"

2.39 MID$

Format: MID$(<string expl>,n[,m])=<string exp2>

where n and m are integer expressions and <string expl> and <string exp2> are string expressions.

Purpose: To replace a portion of one string with another string.

Remarks: The characters in <string expl>, beginning at position n, are replaced by the characters in <string exp2>. The optional m refers to the number of characters from <string exp2> that will be used in the replacement. If m is omitted, all of <string exp2> is used. However, regardless of whether m is omitted or included, the replacement of characters never goes beyond the original length of <string expl>.

Example:
```
10 A$="KANSAS CITY, MO"
20 MID$(A$,14)="KS"
30 PRINT A$
RUN
KANSAS CITY, KS
```

MID$ is also a function that returns a substring of a given string. See Section 3.24.

2.40 NAME

Format: NAME <old filename> AS <new filename>

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example:

```
Ok
NAME "ACCTS" AS "LEDGER"
Ok
```

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Format: NEW

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

2.42 NULL

Format: NULL <integer expression>

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, <integer expression> should be >=3. When tapes are not being punched, <integer expression> should be 0 or 1 for Teletypes and Teletype-compatible terminal screens. <integer expression> should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
     .
     .
     .
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example: 10 ON ERROR GOTO 1000

2.44 ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

ON <expression> GOSUB <list of line numbers>

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example:
```
100 ON L-1 GOTO 150,300,320,390
```

2.45 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O	specifies sequential output mode
I	specifies sequential input mode
R	specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example:
```
10 OPEN "I",2,"INVEN"
```

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.46 OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Purpose: To declare the minimum value for array subscripts.

Remarks: The default base is 0. If the statement

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

2.47 OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example: 100 OUT 32,100

2.48 POKE

Format: POKE I,J
where I and J are integer expressions

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. I must be in the range 0 to 65536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example:
```
10 POKE &H5A00,&HFF
```

2.49 PRINT

Format: PRINT [<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1:

```
10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
 10             0             -25             3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2:

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3:

```
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Format: PRINT USING <string exp>;<list of expressions>

Purpose: To print strings or numbers using a specified format.

Remarks and Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is lonnger than the string, the string will be left-justified in the field and padded with spaces on the right. Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##    ";10.2,5.3,66.789,.234
10.20     5.30    66.79     0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

+ A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
-68.95    +2.40    +55.60    -0.90

PRINT USING "##.##-    ";-68.95,22.449,-7.01
68.95-    22.45     7.01-
```

****** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#    ";12.39,-0.9,765.1
*12.4    *-0.9    765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

**$ The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (__) format.

```
PRINT USING "####,.##";1234.5
1,234.50

PRINT USING "####.##,";1234.5
1234.50,
```

^^^^ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "##.##^^^^";234.56
 2.35E+02

PRINT USING ".####^^^^-";888888
 .8889E+06

PRINT USING "+.##^^^^";123
+.12E+03
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "##.##";111.22
%111.22

PRINT USING ".##";.999
%1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# AND PRINT# USING

Format: PRINT#<filenumber>,[USING<string exp>;]<list of exps>

Purpose: To write data to a sequential disk file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described in Section 2.50, PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT#1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1". The statement

```
PRINT#1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

```
PRINT#1,A$;",";B$
```

The image written to disk is

```
CAMERA,93604-1
```

which can be read back into two string

variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$=" 93604-1". The statement

```
PRINT#1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "AUTOMATIC 93604-1" to B$. To separate these strings properly on the disk, write double quotes to the disk image using CHR$(34). The statement

```
PRINT#1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement

```
INPUT#1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A$ and " 93604-1" to B$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

```
PRINT#1,USING"$$###.##,";J;K;L
```

For more examples using PRINT#, see Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

See also WRITE#, Section 2.68.

2.52 PUT

Format: PUT [#]<file number>[,<record number>]

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.53 RANDOMIZE

Format: RANDOMIZE [<expression>]

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3  (user
types 3)
 .88598  .484668  .586328  .119426  .709225
Ok
RUN
Random Number Seed (-32768 to 32767)? 4 (user
types 4 for new sequence)
 .803506  .162462  .929364  .292443  .322921
Ok
RUN
Random Number Seed (-32768 to 32767)? 3 (same
sequence as first RUN)
 .88598  .484668  .586328  .119426  .709225
Ok
```

2.54 READ

Format: READ <list of variables>

Purpose: To read values from a DATA statement and assign them to variables. (See DATA, Section 2.10.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statment. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.57)

Example 1:

```
.
.
.
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
.
.
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
LIST
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY          STATE          ZIP
DENVER,       COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Format: REM <remark>

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

WARNING: Do not use this in a data statement as it would be considered legal data.

Example:

```
    .
    .
    .
120 REM CALCULATE AVERAGE VELOCITY
130 FOR I=1 TO 20
140 SUM=SUM + V(I)
    .
    .
    .
```

or:

```
    .
    .
    .
120 FOR I=1 TO 20     'CALCULATE AVERAGE VELOCITY
130 SUM=SUM+V(I)
140 NEXT I
    .
    .
    .
```

2.56 RENUM

Format: RENUM [[<new number>][,[<old number>][,<increment>]]]

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples:

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 300,,50	Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.
RENUM 1000,900,20	Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.57 RESTORE

Format: RESTORE [<line number>]

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
    .
    .
    .
```

2.58 RESUME

Formats:

```
RESUME

RESUME 0

RESUME NEXT

RESUME <line number>
```

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <line number>	Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example:

```
10 ON ERROR GOTO 900
  .
  .
  .
900 IF (ERR=230)AND(ERL=90) THEN PRINT "TRY
AGAIN":RESUME 80
  .
  .
  .
```

2.59 RUN

Format 1: RUN [<line number>]

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC always returns to command level after a RUN is executed.

Example: RUN

Format 2: RUN <filename>[,R]

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVEd. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

NOTE: The Microsoft BASIC Compiler supports the RUN and RUN <line number> forms of the RUN statement. The Microsoft BASIC Compiler does not support the "R" option with RUN. If you want this feature, the CHAIN statement should be used.

2.60 SAVE

Format: SAVE <filename>[,A | ,P]

Purpose: To save a program file on disk.

Remarks: <filename> is a quoted string that conforms to your operating system's requirements for filenames. (Your operating system may append a default filename extension if one was not supplied in the SAVE command. Refer to the Microsoft BASIC User's Guide, PART III for information about possible filename extensions under your operating system.) If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires and ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

Examples:
```
SAVE"COM2",A
SAVE"PROG",P
```

See also PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

2.61 STOP

Format: STOP

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

```
Break in line nnnnn
```

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.62 SWAP

Format: SWAP <variable>,<variable>

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPped (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example:

```
LIST
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
 ONE FOR ALL
 ALL FOR ONE
Ok
```

2.63 TRON/TROFF

Format: TRON

TROFF

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example:

```
TRON
Ok
LIST
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
Ok
RUN
[10][20][30][40] 1  10  20
[50][60][30][40] 2  20  30
[50][60][70]
Ok
TROFF
Ok
```

2.64 WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example:
```
100 WAIT 32,2
```

2.65 WHILE...WEND

Format:

```
WHILE <expression>
  .
  .
[<loop statements>]
  .
  .
WEND
```

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example:

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115         FLIPS=0
120         FOR I=1 TO J-1
130                 IF A$(I)>A$(I+1) THEN
                            SWAP A$(I),A$(I+1):FLIPS=1
140         NEXT I
150 WEND
```

2.66 WIDTH

Format: WIDTH [LPRINT] <integer expression>

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If <integer expression> is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example:

```
10 PRINT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNOPQR
STUVWXYZ
Ok
```

2.67 WRITE

Format: WRITE[<list of expressions>]

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of te expressions are output at thee terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.49.

Example:

```
10 A=80:B=90:C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
 80, 90,"THAT'S ALL"
Ok
```

2.68 WRITE#

Format: WRITE#<file number>,<list of expressions>

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE#1,A$,B$
```

writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent INPUT# statement, such as:

```
INPUT#1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

CHAPTER 3

MICROSOFT BASIC FUNCTIONS

The intrinsic functions provided by Microsoft BASIC are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC will round the fractional portion and use the resulting integer.

NOTE

With the Microsoft BASIC interpreter, only integer and single precision results are returned by functions. Double precision functions are supported only by the Microsoft BASIC Compiler.

3.1 ABS

Format: ABS(X)

Action: Returns the absolute value of the expression X.

Example:
```
PRINT ABS(7*(-5))
 35
Ok
```

3.2 ASC

Format: ASC(X$)

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix D for ASCII codes.) If X$ is null, an "Illegal function call" error is returned.

Example:
```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

See the CHR$ function for ASCII-to-string conversion.

3.3 ATN

Format: ATN(X)

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:
```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
 1.24905
Ok
```

3.4 CDBL

Format: CDBL(X)

Action: Converts X to a double precision number.

Example:
```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
 454.67  454.6700134277344
Ok
```

3.5 CHR$

Format: CHR$(I)

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix D.) CHR$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR$(7)) as a preface to an error message, or a form feed could be sent (CHR$(12)) to clear a terminal screen and return the cursor to the home position.

Example:
```
PRINT CHR$(66)
B
Ok
```
See the ASC function for ASCII-to-numeric conversion.

3.6 CINT

Format: CINT(X)

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example:
```
PRINT CINT(45.67)
 46
Ok
```
See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

3.7 COS

Format: COS(X)

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:
```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.8 CSNG

Format: CSNG(X)

Action: Converts X to a single precision number.

Example:
```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

3.9 CVI, CVS, CVD

Format: CVI(<2-byte string>)
CVS(<4-byte string>)
CVD(<8-byte string>)

Action: Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.
.
.
70 FIELD #1,4 AS N$, 12 AS B$, ...
80 GET #1
90 Y=CVS(N$)
.
.
.
```

See also MKI$, MKS$, MKD$, Section 3.25 and PART II, Chapter 3, Microsoft BASIC Disk I/O, of the Microsoft BASIC User's Guide.

3.10 EOF

Format: EOF(<file number>)

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"
20 C=0
30 IF EOF(1) THEN 100
40 INPUT #1,M(C)
50 C=C+1:GOTO 30
.
.
.
```

3.11 EXP

Format: EXP(X)

Action: Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 X = 5
20 PRINT EXP (X-1)
RUN
 54.5982
Ok
```

3.12 FIX

Format: FIX(X)

Action: Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.

Examples:
```
PRINT FIX(58.75)
 58
Ok

PRINT FIX(-58.75)
-58
Ok
```

3.13 FRE

Format: FRE(0)
FRE(X$)

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example:
```
PRINT FRE(0)
 14542
Ok
```

3.14 HEX$

Format: HEX$(X)

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX$(X) is evaluated.

Example:
```
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT$ function for octal conversion.

3.15 INKEY$

Format: INKEY$

Action: Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through tto the program except for Control-C, which terminates the program. (With the Microsoft BASIC Compiler, Control-C is also passed through to the program.)

Example:

```
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A$=INKEY$ : IF LEN(A$)=0 THEN 1060
1040 IF ASC(A$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE$=RESPONSE$+A$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN
```

3.16 INP

Format: INP(I)

Action: Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.

Example:

```
100 A=INP(255)
```

3.17 INPUT$

Format: INPUT$(X[,[#]Y])

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT$ function.

Example 1:

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN"I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

Example 2:

```
  .
  .
  .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
  .
  .
  .
```

3.18 INSTR

Format: INSTR([I,]X$,Y$)

Action: Searches for the first occurrence of string Y$ in X$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X$) or if X$ is null or if Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions or string literals.

Example:
```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)
RUN
 2  6
Ok
```

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.19 INT

Format: INT(X)

Action: Returns the largest integer <=X.

Examples:
```
PRINT INT(99.89)
 99
Ok

PRINT INT(-12.11)
-13
Ok
```

See the FIX and CINT functions which also return integer values.

3.20 LEFT$

Format: LEFT$(X$,I)

Action: Returns a string comprised of the leftmost I characters of X$. I must be in the range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be returned. If I=0, the null string (length zero) is returned.

Example:
```
10 A$ = "BASIC"
20 B$ = LEFT$(A$,5)
30 PRINT B$
BASIC
Ok
```

Also see the MID$ and RIGHT$ functions.

3.21 LEN

Format: LEN(X$)

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example:
```
10 X$ = "PORTLAND, OREGON"
20 PRINT LEN(X$)
 16
Ok
```

3.22 LOC

Format: LOC(<file number>)

Action: With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example:
```
200 IF LOC(1)>50 THEN STOP
```

3.23 LOG

Format: LOG(X)

Action: Returns the natural logarithm of X. X must be greater than zero.

Example:
```
PRINT LOG(45/7)
 1.86075
Ok
```

3.24 LPOS

Format: LPOS(X)

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example:
```
100 IF LPOS(X)>60 THEN LPRINT CHR$(13)
```

3.25 MID$

Format: MID$(X$,I[,J])

Action: Returns a string of length J characters from X$ beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example:
```
LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok
```

Also see the LEFT$ and RIGHT$ functions.

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.26 MKI$, MKS$, MKD$

Format:
```
MKI$(<integer expression>)
MKS$(<single precision expression>)
MKD$(<double precision expression>)
```

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

Example:
```
90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
 .
 .
```

See also CVI, CVS, CVD, Section 3.9 and Microsoft BASIC Disk I/O, in the Microsoft BASIC User's Guide.

3.27 OCT$

Format: OCT$(X)

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:
```
PRINT OCT$(24)
 30
Ok
```

See the HEX$ function for hexadecimal conversion.

3.28 PEEK

Format: PEEK(I)

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, Section 2.48.

Example:
```
A=PEEK(&H5A00)
```

3.29 POS

Format: POS(I)

Action: Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example:
```
IF POS(X)>60 THEN PRINT CHR$(13)
```

Also see the LPOS function.

3.30 RIGHT$

Format: RIGHT$(X$,I)

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If I=0, the null string (length zero) is returned.

Example:
```
10 A$="DISK BASIC"
20 PRINT RIGHT$(A$,8)
RUN
BASIC
Ok
```

Also see the MID$ and LEFT$ functions.

3.31 RND

Format: RND[(X)]

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:
```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

3.32 SGN

Format: SGN(X)

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

3.33 SIN

Format: SIN(X)

Action: Returns the sine of X in radians. SIN(X) is calculated in single precision. COS(X)=SIN(X+3.14159/2).

Example:
```
PRINT SIN(1.5)
 .997495
Ok
```

3.34 SPACE$

Format: SPACE$(X)

Action: Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.

Example:
```
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
  1
   2
    3
     4
      5
Ok
```

Also see the SPC function.

3.35 SPC

Format: SPC(I)

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A';' is assumed to follow the SPC(I) command.

Example:
```
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

Also see the SPACE$ function.

3.36 SQR

Format: SQR(X)

Action: Returns the square root of X. X must be >=0.

Example:
```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.16228
 15            3.87298
 20            4.47214
 25            5
Ok
```

3.37 STR$

Format: STR$(X)

Action: Returns a string representation of the value of X.

Example:
```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
    .
    .
    .
```

Also see the VAL function.

3.38 STRING$

Formats:
```
STRING$(I,J)
STRING$(I,X$)
```

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X$.

Example:
```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

3.39 TAB

Format: TAB(I)

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:
```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                      AMOUNT

G. T. JONES               $25.00
Ok
```

3.40 TAN

Format: TAN(X)

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example:
```
10 Y = Q*TAN(X)/2
```

3.41 USR

Format : USR[<digit>](X)

Action: Calls the user's assembly language subroutine with the argument X. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Assembly Language Subroutines, in the Microsoft BASIC User's Guide.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
   .
   .
   .
```

3.42 VAL

Format: VAL(X$)

Action: Returns the numerical value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

VAL(" -3)

returns -3.

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
   .
   .
   .
```

See the STR$ function for numeric to string conversion.

3.43 VARPTR

Format 1: VARPTR(<variable name>)

Format 2: VARPTR(#<file number>)

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2: For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

Example:

```
100 X=USR(VARPTR(Y))
```

APPENDIX A

Summary of Error Codes and Error Messages

Code	Number	Message
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent

5. a call to a USR function for which the starting address has not yet been given

6. an improper argument to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO.

OV 6 Overflow
The result of a calculation is too large to be represented in Microsoft BASIC's number format. If underflow occurs, the result is zero and execution continues without an error.

OM 7 Out of memory
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

UL 8 Undefined line
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.

BS 9 Subscript out of range
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

DD 10 Redimensioned array
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.

/0 11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

ID 12 Illegal direct
A statement that is illegal in direct mode is entered as a direct mode command.

TM 13 Type mismatch
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument

or vice versa.

OS 14 Out of string space
String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.

LS 15 String too long
An attempt is made to create a string more than 255 characters long.

ST 16 String formula too complex
A string expression is too long or too complex. The expression should be broken into smaller expressions.

CN 17 Can't continue
An attempt is made to continue a program that:

1. has halted due to an error,
2. has been modified during a break in execution, or
3. does not exist.

UF 18 Undefined user function
A USR function is called before the function definition (DEF statement) is given.

19 No RESUME
An error trapping routine is entered but contains no RESUME statement.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

22 Missing operand
An expression contains an operator with no operand following it.

23 Line buffer overflow
An attempt is made to input a line that has too many characters.

26 FOR without NEXT

A FOR was encountered without a matching NEXT.

29 WHILE without WEND
A WHILE statement does not have a matching WEND.

30 WEND without WHILE
A WEND was encountered without a matching WHILE.

Disk Errors

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Microsoft BASIC. Report to Microsoft the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is exeucted after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

66 Direct statement in file
A direct statement is encountered while LOADing an ASCII-format file. The LOAD is terminated.

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX B

Mathematical Functions

Derived Functions

Functions that are not intrinsic to Microsoft BASIC may be calculated as follows.

Function	Microsoft BASIC Equivalent
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX C

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

INDEX

ABS 3-2
Addition 1-10
ALL 2-4, 2-9
Arctangent 3-3
Array variables 1-7, 2-9, 2-19
Arrays 1-7, 2-7, 2-12, 2-25
ASC 3-2
ASCII codes 3-2, 3-4
ASCII format 2-4, 2-50, 2-78
Assembly language subroutines 2-3, 2-17, 2-60, 3-23 to 3-24
ATN 3-3
AUTO 1-2, 2-2

Boolean operators 1-12

CALL 2-3
Carriage return . . . . . 1-3, 2-37, 2-42 to 2-43, 2-84 to 2-86
Cassette tape 2-7, 2-12
CDBL 3-3
CHAIN 2-4, 2-9
Character set 1-3
CHR$ 3-4
CINT 3-4
CLEAR 2-6
CLOAD 2-7
CLOAD* 2-7
CLOAD? 2-7
CLOSE 2-8
Command level 1-1
COMMON 2-4, 2-9
Concatenation 1-15
Constants 1-4
CONT 2-11, 2-42
Control characters 1-4
Control-A 2-23
COS 3-5
CSAVE 2-12
CSAVE* 2-12
CSNG 3-5
CVD 3-6
CVI 3-6
CVS 3-6

DATA 2-13, 2-75
DEF FN 2-14
DEF USR 2-17, 3-23
DEFDBL 1-7, 2-16
DEFINT 1-7, 2-16
DEFSNG 1-7, 2-16

DEFSTR 1-7, 2-16
DELETE 1-2, 2-4, 2-18
DIM 2-19
Direct mode 1-1, 2-35, 2-55
Division 1-10
Double precision 1-5, 2-16, 2-61, 3-3

EDIT 1-2, 2-20
Edit mode 1-4, 2-20
END 2-8, 2-11, 2-24, 2-33
EO 3-6
ERASE 2-25
ERL 2-26
ERR 2-26
ERROR 2-27
Error codes . . . . . 1-16, 2-26 to 2-27, A-1
Error messages 1-16, A-1
Error trapping . . . . . 2-26 to 2-27, 2-55, 2-76
Escape 1-3, 2-20
EXP 3-7
Exponentiation . . . . . 1-10 to 1-11
Expressions 1-9

FIELD 2-29
FIX 3-7
FOR...NEXT 2-30
FRE 3-8
Functions 1-14, 2-14, 3-1, B-1

GET 2-29, 2-32
GOSUB 2-33
GOTO . . . . . 2-33 to 2-34

HEX$ 3-8
Hexadecimal 1-5, 3-8

IF...GOto 2-35
IF...THEN 2-26, 2-35
IF...THEN...ELSe 2-35
IN 3-12
Indirect mode 1-1
INKEY$ 3-9
INP 3-9
INPUT 2-11, 2-29, 2-37
INPUT# 2-39
INPUT$ 3-10
INSTR 3-11
INT 3-7
Integer 3-4, 3-7, 3-12
Integer division 1-11
ISIS-II 2-77

KILL 2-40

LEFT$ 3-12

LEN 3-13
LET 2-29, 2-41
Line feed . . . . . 1-2, 2-37, 2-42 to 2-43, 2-85 to 2-86
LINE INPUT 2-42
LINE INPUT# 2-43
Line numbers . . . . . 1-1 to 1-2, 2-2, 2-74
Line printer 2-46, 2-48, 2-84, 3-14
Lines 1-1
LIST 1-2, 2-44
LLIST 2-46
LOAD 2-47, 2-78
LOC 3-13
LOG 3-14
Logical operators 1-12
Loops 2-30, 2-83
LPOS 2-84, 3-14
LPRINT 2-48, 2-84
LPRINT Using 2-48
LSET 2-49

MERGE 2-4, 2-50
MID$ 2-51, 3-15
MKD$ 3-15
MKI$ 3-15
MKS$ 3-15
MOD OPERATOR 1-11
Modulus arithmetic 1-11
Multiplication 1-10

NAME 2-52
Negation 1-10
NEW 2-8, 2-53
NULL 2-54
Numeric constants 1-4
Numeric variables 1-7

OCT$ 3-16
Octal 1-5, 3-16
ON ERROR GOTO 2-55
ON...GOSUB 2-56
ON...GOTO 2-56
OPEN 2-8, 2-29, 2-57
Operators . . . . . 1-9, 1-11 to 1-13, 1-15
OPTION BASE 2-58
OUT 2-59
Overflow 1-11, 3-7, 3-22
Overlay 2-4

Paper tape 2-54
PEEK 2-60, 3-16
POKE 2-60, 3-16
POS 2-84, 3-17
PRINT 2-61
PRINT USING 2-63

PRINT# 2-67
PRINT# USING 2-67
Protected files 2-78
PUT 2-29, 2-69

Random files 2-29, 2-32, 2-40, 2-49, 2-57, 2-69, 3-13, 3-15
Random numbers 2-70, 3-18
RANDOMIZE 2-70, 3-18
READ 2-71, 2-75
Relational operators 1-11
REM 2-73
RENUM 2-4, 2-26, 2-74
RESTORE 2-75
RESUME 2-76
RETURN 2-33
RIGHT$ 3-17
RND 2-70, 3-18
RSET 2-49
Rubout 1-3, 1-15, 2-21
RUN . . . . . 2-77 to 2-78

SAVE . . . . . 2-47, 2-77 to 2-78
Sequential files . . . . . 2-39 to 2-40, 2-43, 2-57, 2-67, 2-86, 3-6, 3-13
SGN 3-18
SIN 3-19
Single precision 1-5, 2-16, 2-61, 3-5
Space Requirements for variables 1-8
SPACE$ 3-19
SPC 3-20
SQR 3-20
STOP 2-11, 2-24, 2-33, 2-79
STR$ 3-21
String constants 1-4
String functions . . . . . 3-6, 3-11 to 3-13, 3-15, 3-17, 3-21, 3-23
String operators 1-15
String space 2-6, 3-8
String variables . . . . . 1-7, 2-16, 2-42 to 2-43
STRING$ 3-21
Subroutines 2-3, 2-33, 2-56
Subscripts 1-7, 2-19, 2-58
Subtraction 1-10
SWAP 2-80

TAB 3-22
Tab . . . . . 1-3 to 1-4
TAN 3-22
TROFF 2-81
TRON 2-81

USR 2-17, 3-23

VAL 3-23

Variables 1-6
VARPTR 3-24

WAIT 2-82
WEND 2-83
WHILE 2-83
WIDTH 2-84
WIDTH LPRINT 2-84
WRITE 2-85
WRITE# 2-86

MICROSOFT GW-BASIC ADDENDUM

Language Feature Extensions to Standard BASIC-86 Version 5.0

for the iAPX 8088/8086 Microprocessors

CHAPTER 1

GW-BASIC LANGUAGE FEATURES FOR THE iAPX 88/86

NOTE
The time saving benefit of the Full Screen Editor during program development cannot be over emphasized. To that end, it is suggested that a sample program be entered and each edit command practiced until it becomes second nature. In the following discussion of edit commands, the term "cursor" refers to the "blinking" line appearing just to the right of the last character typed. This marks the next position at which a character is to be inserted or deleted. The dynamic nature of editing anywhere on the screen makes it difficult to provide clear examples of command usage in printed text; therefore, the best way of getting the "feel" for the editing process is to try editing a few lines while studying the edit commands that follow.

1.1 THE BASIC FULL SCREEN EDITOR

1.1.1 Inputting the BASIC Program

Any line of text typed while BASIC is in Direct Mode will be processed by the Full Screen Editor. Basic is always in Direct Mode after the prompt OK and until a RUN command is given.

Any line of text typed that begins with a numeric character (digit) is considered a Program statement and will be processed in one of four ways:

1. A new line is added to the program. This occurs if the line number is legal (range is 0 thru 65529) and at least one non-blank character follows the line number in the line.

2. An existing line is modified. This occurs if the line number matches the line number of an existing line in the program. This line is replaced with the text of the newly entered line.

3. An existing line is deleted. This occurs if the line number matches the line number of an existing line and the entered line contains ONLY a line number.

4. An error is produced.

   a. If an attempt is made to delete a non-existent line, an "Undefined line number" error message is displayed.

   b. If program memory is exhausted, and a line is added to the program, the error; "Out of Memory" is displayed and the line is not added.

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon (:).

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of 250 characters.

It is possible to extend a logical line over more than one physical Full Screen by using the line-feed key. Typing a line-feed causes subsequent text to start on the next line without entering a carriage return. Entering a carriage return passes the entire logical line to BASIC for storage in the program.

Occasionally, BASIC may return to Direct Mode with the cursor positioned on a line containing a message issued by BASIC such as "OK." When this happens, the line is automatically erased. This is provided as a courtesy to the programmer. If the line were not erased and the programmer typed carriage return, the message would pass to BASIC and a "Syntax Error" would result. BASIC messages are terminated by HEX 'FF' to distinguish them from user text.

1.1.2 Altering Lines with the Editor

Editing existing lines on the Screen is achieved by moving the cursor on the screen to the place requiring change and then performing one of the following five functions:

1. Typing over characters already there.

2. Deleting characters to the left of the cursor.

3. Deleting characters to the right of the cursor.

4. Inserting characters at the cursor while shifting later characters to the right.

5. Adding, or appending, characters to the end of the current logical line.

The Full Screen Editor recognizes 9 special or numeric key-pad keys, the backspace key, the ESC key, and the CTRL key for moving the cursor to a location on the screen, inserting characters, or deleting characters. The keys and their values are:

HEX	DEC	FUNCTIONAL NAME
ØD	13	Carriage Return*
ØA	1Ø	Line Feed*
ØB	11	HOME
1E	30	CURSOR UP
1F	31	CURSOR DOWN
1D	29	CURSOR LEFT
1C	28	CURSOR RIGHT
ØE	14	NEXT WORD
Ø2	Ø2	PREVIOUS WORD
12	18	INS (Insert Mode Toggle)
Ø8	Ø8	BS (Backspace)*
7F	128	DEL (Delete Character)
Ø5	Ø5	CTRL-END (Erase to End Of Line)
ØC	12	CTRL-HOME (Clear Screen)
1B	27	ESC (Erase Line)
15	21	ESC (Erase Line)*
Ø9	Ø9	TAB*
Ø3	Ø3	BREAK (Stop Program)*

* Not numeric keypad

GW-BASIC - DOT NUMERIC KEYPAD			
% Delete Word	X	/ Delete Line	CLR Delete to End of Window
7	8 Cursor Up	9	+
4 Cursor Left	5 Cursor Home	6 Cursor Right	-
1 Move to EOL	2 Cursor Down	3 Delete Word	ENTER Edit Line Where Error Occurred
Ø Toggle Insert/Overstrike		. Delete Character	

DOT NUMERIC KEYPAD (WITH CTRL DEPRESSED)			
% Display Function Keys	X	/	CLR
7	8	9	+
4 Back One Word	5 Clear Window	6 Next Word	-
1 Delete to EOL	2	3	ENTER
Ø		.	

HOME Moves the cursor to the upper left hand corner of the screen.

CTRL-HOME Clears the screen and positions the cursor in the upper left hand corner of the screen.

CURSOR UP Moves the cursor up one line.

CURSOR DOWN Moves the cursor one position down.

CURSOR LEFT Moves the cursor one position left. When the cursor is advanced beyond the left of the screen, it moves to the preceeding line on the right side of the screen.

CURSOR RIGHT Moves the cursor one position right. When the cursor is advanced beyond the right side of the screen, it moves to the left side on the next line on the screen.

NEXT WORD Depressing the CTRL and CURSOR RIGHT keys moves the cursor right to the next word. The next word is defined as the next character to the right of the cursor in the set [A..Z] or [0..9].

PREVIOUS WORD Depressing the CTRL and CURSOR LEFT keys moves the cursor left to the previous word. The previous word is defined as the next character to the left of the cursor in the set A..Z or 0..9.

1 NK* Moves the cursor to the end of the Logical Line. Characters typed from this position are appended to the line.

CTRL-1 NK* Depressing the CTRL and 1 NK keys erases from the current cursor position to the End of the Logical Line. All physical Full Screens are erased until the terminating carriage return is found.

* 1 NK represents the Numeric Keypad with NUM LOCK toggled to OFF (LED not lit).

INS — Toggles Insert Mode. If Insert Mode is off, INS turns it on. If on, then INS turns it off.

Insert Mode is indicated by the blinking cursor blotting the lower half of the character position. In Graphic Modes, the normal cursor covers the whole character position. When Insert Mode is active, only the lower half of the character position is blotted.

When out of Insert Mode, characters typed will replace existing characters on the line.

TAB — When out of Insert Mode, depressing the TAB key moves the cursor over characters until the next tab stop is reached. Tab stops occur every 8 character positions.

When in Insert Mode, characters following the cursor are moved to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding is observed. That is, as characters advance off the right side of the screen, they are inserted from the left on subsequent lines.

When in Insert Mode, depressing the TAB key inserts blanks from the current cursor position to the next tab stop. Line folding is observed as above.

DEL — Deletes one character immediately to the right of the cursor for each depression. All characters to the right of the one deleted are then moved one position left to fill in the one deleted. If a logical line extends beyond one physical line, characters on subsequent lines are moved left one position to fill in the previous space, and the character in the lst column of each subsequent line is moved up to the end of the preceeding line.

BS — Back Space. Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of the cursor are moved left one position. subsequent characters and lines within the current logical line are moved up as with the DEL key.

ESC — When typed anywhere in the line causes the entire logical line to be erased.

BREAK — returns to Direct Mode, without saving any changes that were made to the current line being edited.

1.1.3 Changing a BASIC Program

Modifying existing programs is achieved by displaying program lines on the screen with the LIST statement. List the range of lines to be edited, (See "The LIST statement"). Position the cursor to the line to be edited, modify the line using the keys described in "Altering Lines with the Editor". Type carriage return to store the modified line in the program.

NOTE: A program line is not actually modified within the Basic Program until RETURN is entered. Therefore, when several lines need alteration, it is sometimes easier to move around the screen making corrections to several lines at once, and then, go back to the first line changed and strike RETURN at the beginning of each line and by doing so, store the modified line in the program.

Note that it is not necessary to move the cursor to the end of the logical line before typing the carriage return. The Full Screen Editor remembers where each logical line ends and transfers the whole line even if the carriage return is typed at the beginning of the line.

To truncate a line at the current cursor position, type the CTRL-END keys followed by a carriage return.

SYNTAX ERRORS

When a Syntax Error is encountered during program execution, BASIC automatically enters EDIT at the line that caused the error. For example:

```
10 A = 2$12
RUN
?Syntax Error in 10
10 A = 2$12
```

The Full Screen Editor has displayed the line in error and positioned the cursor under the digit 1. The user moves the cursor right to the dollar sign ($) and changes it to an up-arrow (^), followed by a carriage return. The corrected line is now stored back in the program.

In this example, storing the line back in the program causes all variables to be lost. Had the programmer wanted to examine the contents of some variable before making the change, BREAK would be typed to return to Direct Mode. The variables would be preserved since no program line was changed, and after the programmer was satisfied, the line could be edited and the program re-run.

1.1.4 The EDIT Statement

With the Full Screen Editor, the EDIT statement simply displays the line specified and positions the cursor under the first digit of the line number. The line may then be modified using the keys described in "Altering Lines with the Editor".

Format:

```
EDIT <line number>
EDIT .
```

WHERE:

line number	is the program line number of a line existing in the program. If there is no such line, an "Undefined Line number" error message is displayed.
.	Always gets the last line referenced by an EDIT statement, LIST command, or Error message.

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT ." will enter EDIT at the current line. (The line number symbol "." always refers to the current line).

1.1.5 New Error Messages

24 Device Timeout
Basic did not receive information from an I/O device within a predetermined amount of time. In Level 1 GW-BASIC, this would only occur during a Cassette read or Line Printer write.

25 Device Fault
In Level 1 GW-BASIC, will only occur for the Line Printer when true fault status is returned from the centronics printer interface. Usually indicates a hardware error in the printer or interface card.

57 Device I/O Error
Formerly "Disk I/O Error". Generalized to include all I/O Devices.

1.2 LEVEL 1 GW-BASIC STATEMENT EXTENSIONS

1.2.1 The BEEP Statement

Function:

The BEEP statement sounds the Speaker at 800 Hz for 1/4 second.

Non graphic versions of Microsoft Basic use PRINT CHR$(7) to send an ASCII Bell character. Both BEEP and PRINT CHR$(7); have the same affect in Level 1 GW-BASIC.

Format:

```
BEEP
```

Example:

```
2430 IF X < 20 THEN BEEP 'X is out of range, complain.
```

1.2.2 The SOUND Statement

Function:

The SOUND statement generates sound through the speaker.

Format:

```
SOUND freq,duration
```

WHERE:

freq — is the desired frequency in Hertz. A valid numeric expression returning an unsigned Integer in the range 37 to 32767.

duration — is the desired duration in Clock ticks. A valid numeric expression returning an unsigned Integer in the range 0 to 65535.

Current clock ticks occur 18.2 times per second.

Rules:

1. If the duration is zero, any current SOUND statement that is running is turned off. If no SOUND statement is running, SOUND x,0 has no effect.

Example:

```
2500 SOUND RND*1000+37,2 'Creates random sounds.
```

1.2.3 The BLOAD and BSAVE Statements

Function:

The BLOAD statement allows a file to be loaded anywhere in user memory.

The BSAVE statement allows portions of the users memory to be saved on Cassette.

BLOAD and BSAVE are most useful for loading and saving machine language programs. (See "The CALL statement"). However, BLOAD and BSAVE are not restricted to only machine language programs. Any segment may be specified as the source or target for these statements via the DEF SEG statement. This allows the Video Screen buffer to be read or written from cassette. A useful way of saving and displaying graphic images.

1.2.3.1 The BLOAD Statement

Format:

```
BLOAD <file spec> [,<offset>]
```

WHERE:

file spec	Is a valid string expression containing the device and file name. The device must be 4 characters in length. The file name may be 1 to 8 characters in length.
offset	Is a valid numeric expression returning an unsigned Integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement at which loading is to start.

Action:

The Cassette motor is turned on and the cassette is searched for the named file.

If the BLOAD statement is entered in direct mode, the file names on the tape will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message "Skipped." for the files not matching the named file, and "Found." when the named file is found. Types of files and their letter are:

.B For Binary Basic Programs.
.P For Protected Binary Basic Programs.
.A For Ascii Basic Programs.
.M For Memory Image files.
.D For Data Files.

To see what files are on a cassette tape, enter; LOAD"FOO" or some other name that is known not to be on the tape. All file names will then be displayed.

Note that BREAK may be typed at any time during BLOAD or LOAD. between files or after a time-out period, Basic will exit the search and return to Direct Mode. Previous memory contents remain unchanged.

If the BLOAD command is executed in a Basic program, the file names skipped and found are not displayed on the Screen.

Rules:

1. If device is omitted, CAS1: is assumed. CAS1: is the only allowable device for BLOAD in Level 1 GW-BASIC.

2. If the device identifier (CAS1:) is omitted and the filename is less than 1 character or greater than 8 characters long, a "Bad File Name" error is issued and the load aborted.

3. If offset is omitted, the offset specified at BSAVE is assumed. The file loads to the same location from which it was saved.

4. If offset is specified, execute a DEF SEG statement before BLOAD. When offset is given, BASIC assumes the user wants to BLOAD at an address other than the one saved. BASIC uses the last known DEF SEG address.

5. CAUTION: BLOAD does not perform an address range check. That is, it is possible to BLOAD anywhere in memory. Do not BLOAD over BASIC's stack, the BASIC program, or BASIC's variable area.

Example:

```
10 "Load a machine language program into memory at
     60:F000

20 DEF SEG                      'Restore Segment to
                                 Basics's DS.
30 BLOAD "PROG1",&HF000         'Load PROG1 into the DS.

10 'load the screen buffer from tape..

20 DEF SEG= &HB800              'Point segment at screen
                                 buffer.
30 BLOAD"PICTURE",0             'Load file PICTURE into the
                                 screen buffer.
```

Note that the DEF SEG statement in 20 and the offset of 0 in 30 guarantees that the correct address is used.

The BSAVE example in the next section illustrates how PICTURE was saved.

1.2.3.2 The BSAVE Statement

Format:

BSAVE <file spec>,<offset>,<length>

WHERE:

file spec	Is a valid string expression containing the device and file name. The device must be 4 characters in length. The file name may be 1 to 8 characters in length.
offset	Is a valid numeric expression returning an unsigned Integer in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG to start saving from.
length	Is a valid numeric expression returning an unsigned Integer in the range 1 to 65535. This is the length of the memory image to be saved.

Action:

The Cassette motor is turned on and the memory image file is immediately written to tape.

Rules:

1. If filename is less than 1 character or greater than 8 characters long, a "Bad File Name" error is issued and the save is aborted.

2. If offset is omitted, a "Bad File Name" error is issued and the save is aborted. A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

3. If length is omitted, a "Bad File Name" error is issued and the save is aborted.

Example:

```
10 'Save the screen buffer on tape..

DEF SEG= &HB800           'Point segment at screen buffer.
BSAVE"PICTURE",0,16384    'Save screen buffer in file PICTURE.
```

The DEF SEG statement must be used to set up the Segment address to the start of the screen buffer. Offset of 0 and length 16384 specifies that the entire 16K screen buffer is to be saved.

1.2.4 The CALL Statement

Function:

The CALL statement is the recommended way of interfacing 8086 machine language programs with BASIC. It is further suggested that the old style user call (x=USR(n)) not be used.

Format:

CALL <variable name> [(<argument list>)]

WHERE:

variable name — contains the address that is the starting point in memory of the subroutine being CALLed.

argument list — contains the variables or constants, separated by commas, that are to be passed to the routine.

Action:

The CALL statement conforms to the INTEL PL/M-86 calling conventions outlined in Chapter 9 of the INTEL PL/M-86 Compiler User's Manual. BASIC follows the rules described for the MEDIUM case.

Invocation of the CALL statement causes the following to occur:

1. For each parameter in the argument list; the 2 byte offset into the DS of the parameter's location is pushed onto the stack.
2. The return address Code segment [CS], and offset are pushed onto the Stack.
3. Control is transferred to the user's routine via the segment address given in the last DEF SEG statement, and offset given in <variable name>.

The user's routine now has control. Parameters may be referenced by moving the Stack pointer [SP] to the Base Pointer [BP] and adding a positive offset to [BP].

Rules:

1. The CALLed routine may destroy any registers.

2. The CALLed program MUST know how many parameters were passed. Parameters are referenced via a positive offset being added to [BP]. (Assuming the called routine moved the current stack pointer into BP, ie: MOV BP,SP).

   That is, the location of p1 is at 8[BP], p2 is at 6[BP], p3 is at 4[BP], ...etc.

   The CALLed routine must do a RET <n> where <n> is the number of parameters in the argument list *2. This is necessary in order to adjust the stack to the point at the start of the calling sequence.

3. Values are returned to BASIC by including the variable name which will receive the result in the argument list.

4. If the argument is a string, the parameter's offset points to 3 bytes called the "String Descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

   CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add +"" to the string literal in the program. Example:

   ```
   20 A$ = "BASIC"+""
   ```

   This will force the string literal to be copied into string space. Now the string may be modified without affecting the program.

5. Strings may be altered by user routines but the length MUST NOT be changed! BASIC cannot correctly erase strings if their lengths are modified by external routines.

Example:

```
100 DEF SEG=&H8000
110 FOO=0
120 CALL FOO(A,B$,C)
      .
      .
      .
```

Line 100 sets the segment to 8000 Hex. FOO is set to zero so that the call to FOO will execute the subroutine at location 80000H.

The following sequence of 8086 assembly language demonstrates access of the parameters passed and storing a return result in the variable 'C'.

```
MOV  BP,SP      ;Get current Stack posn in BP.
MOV  BX,6[BP]   ;Get address of B$ dope.
MOV  CL,[BX]    ;Get length of B$ in CL.
MOV  DX,1[BX]   ;Get addr of B$ text in DX.
.
.
.
MOV  SI,8[BP]   ;Get address of 'A' in SI.
MOV  DI,4[BP]   ;Get pointer to 'C' in DI.
MOVS WORD       ;Store variable 'A' in 'C'.
RET  6          ;Restore Stack, return.
```

BEWARE!, the called program must know the variable type for numeric parameters passed. In the above example, the instruction; MOVS WORD will copy only 2 bytes. This is fine if variables A and C are Integer. We would have to copy 4 bytes if they were Single Precision and copy 8 bytes if they were Double Precision.

1.2.5 The COLOR Statement

Function:

The COLOR statement selects the Foreground, Background, and Border Screen display colors.

Syntax depends upon the current mode as set by the SCREEN statement.

For Alpha modes:
Foreground: set 1 of 16 colors, Blink On/Off
Background: set 1 of 16 colors
Border: set 1 of 16 colors

1.2.5.1 The COLOR Statement in Alpha Modes

Format:

For Alpha Modes:
COLOR [Foreground] [, [Background] [,border]]

WHERE:

Foreground :=	Foreground (for character color). An unsigned Integer in the range 0-31.
Background :=	Background Color. An unsigned Integer in the range 0-15.
Border :=	Parameter not used.

In alphanumeric modes, the Foreground and Background colors are combined to generate a character attribute. The possible attributes are:

- Blinking
- Inverse Video
- Highlight
- Underscore
- Overscore

There are some restrictions that apply to combinations of attributes. The Inverse Video, Overscore, and Underscore attributes are mutually exclusive.

The following table displays the attributes resulting from various combinations of Foreground and Background colors.

FOREGROUND COLOR	BACKGROUND COLOR 0	1	2	3	4	5	6	7
0	*							I
1	U	U	U	U	U	U	U	U
2								
3								
4								
5								
6	O	O	O	O	O	O	O	O
7								
8	*	H	H	H	H	H	H	HI
9	HU	HU	HU	HU	HU	HU	HU	HU
10	H	H	H	H	H	H	H	H
11	H	H	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H
13	H	H	H	H	H	H	H	H
14	HO	HO	HO	HO	HO	HO	HO	HO
15	H	H	H	H	H	H	H	H
16	*	B	B	B	B	B	B	BI
17	BU	BU	BU	BU	BU	BU	BU	BU
18	B	B	B	B	B	B	B	B
19	B	B	B	B	B	B	B	B
20	B	B	B	B	B	B	B	B
21	B	B	B	B	B	B	B	B
22	BO	BO	BO	BO	BO	BO	BO	BO
23	B	B	B	B	B	B	B	B
24	*	BH	BH	BH	BH	BH	BH	BHI
25	BHU	BHU	BHU	BHU	BHU	BHU	BHU	BHU
26	BH	BH	BH	BH	BH	BH	BH	BH
27	BH	BH	BH	BH	BH	BH	BH	BH
28	BH	BH	BH	BH	BH	BH	BH	BH
29	BH	BH	BH	BH	BH	BH	BH	BH
30	BHO	BHO	BHO	BHO	BHO	BHO	BHO	BHO
31	BH	BH	BH	BH	BH	BH	BH	BH
32	BH	BH	BH	BH	BH	BH	BH	BH

Where:

B := Blinking
H := High Intensity
I := Inverse Video
O := Overscore
U := Underline
* := Invisible

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" error. Previous values are retained.

2. For Foreground Color, values 0 through 15 select Color. Values 16 through 31 select colors 0 through 15, and cause characters displayed to blink.

3. Foreground Color may equal Background Color. This has the effect of making any character displayed invisible. Changing the Foreground or Background color will make the characters visible again.

4. Any parameter may be omitted. Omitted parameters assume the old value.

5. The COLOR statement may not end with a comma (,). That is, 'COLOR ,7,' is illegal and will result in a "Syntax Error."

1.2.6 The CLS Statement

Function:

The CLS statement erases the Current Active Screen Page. (See "The SCREEN Statement.")

Format:

CLS

Rules:

1. If the Screen is in Alpha Mode, the active page is cleared to the currently selected Background Color. (See "The COLOR Statement.")

2. If the Screen is in Graphics medium or in Hi-res mode, the entire Screen buffer is cleared to Black.

3. The Screen may also be cleared by depressing the CTRL-L or CTRL-HOME keys.

4. NOTE: The SCREEN and WIDTH statements will force a Screen clear if the resultant Screen mode created is different than the mode currently in force.

Example:

```
1 CLS      'Clears the screen.
```

1.2.7 The CSRLIN Function

Function:

The CSRLIN Function returns the current line (or Row) position of the cursor.

Format:

```
x = CSRLIN
```

WHERE:

x is a numeric variable receiving the value returned. The value returned will be in the range 1 to 24.

x = POS(0) will return the column locations of the cursor.

Example:

```
10 Y = CSRLIN  'Record current line.
20 X = POS(0)  'Record current column.
30 LOCATE 24,1 :PRINT "HELLO" 'Print HELLO on last line.
40 LOCATE Y,X  'Restore position to old line, column.
```

1.2.8 The DEF SEG Statement

Function:

The DEF SEG statement assigns the current value to be used by a subsequent BLOAD, BSAVE, CALL, or user defined function call.

Format:

DEF SEG [=<address>]

WHERE:

address is a valid numeric expression returning an unsigned Integer in the range 0 to 65535.

Action:

The address specified is saved for use as the segment required by the BLOAD, BSAVE, PEEK, POKE and CALL statements.

Rules:

1. Any value entered outside of this range will result in an "Illegal Function Call" Error. The previous value is retained.

2. If the address option is omitted, the segment to be used is set to Basic's Data Segment. This is the initial default value.

3. If the address option is given, it should be a value based upon a 16 byte boundary. For the BLOAD, BSAVE, PEEK, POKE, or CALL statements, the value is shifted left 4 bits to form the Code Segment address for the subsiquent call instruction. Basic does not perform additional checking to assure that the resultant segment + offset value is valid.

4. NOTE: DEF and SEG MUST be separated by a space! Otherwise, Basic would interpret the statement; DEFSEG=100 to mean: "assign the value 100 to the variable DEFSEG".

Example:

```
10 DEF SEG=&HB800    'Set segment to Screen buffer.
20 DEF SEG           'Restore segment to Basic's DS.
```

1.2.9 The KEY Statement

Function:

The KEY statement allows Function keys to be designated "Soft Keys". Any one or all of the ten Special Function Keys may be assigned a 15 byte string which, when the Key is depressed, will be input to Basic.

Initially, the Soft Keys are assigned the following values:

F1 - LIST	F2 - RUN
F3 - LOAD	F4 - SAVE
F5 - CONT	F6 - ,"LPT1:"
F7 - TRON	F8 - TROFF
F9 - KEY	F10 - SCREEN 0,0,0

Format:

```
KEY <key number>,<string expression>
KEY LIST
KEY ON
KEY OFF
```

WHERE:

key number	is the Key number. An Expression returning an unsigned Integer in the range 1 to 10.
string expression	is the Key assignment text. Any valid string expression.

Action:

KEY ON	This is the initial setting. Causes the Key values to be displayed on the 25th Line. When the Width is 40, 5 of the 10 Soft Keys are displayed. When the width is 80, all 10 are displayed. In either width, only the first 7 characters of each value are displayed.
KEY OFF	Erases the Soft Key display from the 25th line.
KEY LIST	Lists all 10 Soft Key values on the screen. All 15 characters of each value are displayed.

KEY <key number>,<string expression>
Assigns the string expression to the Soft Key specified (1 to 10).

Rules:

1. If the value returned for <key number> is not in the range 1 to 10, an "Illegal Function Call" Error is taken. The previous Key string assignment is retained.

2. The Key assignment string may be 1 to 15 characters in length. If the string is longer than 15 characters, the first 15 characters are assigned.

3. Assigning a null string (string of length 0) to a Soft Key disables the Function Key as a Soft Key.

4. When a Soft Key is assigned, the INKEY$ function returns one character of the soft key string per invocation. If the Soft Key is disabled, INKEY$ returns a string of length 2. The first character is binary zero, the second is the Key Scan Code.

Example:

50 KEY ON	Display the Soft Keys on the 25th Line.
200 KEY OFF	Erase Soft Key display.
10 KEY 1,"MENU"+CHR$(13)	Assigns the string 'MENU'<carriage return> to soft key 1. Such assignments might be used for rapid data entry. This example might be used in a program to select a menu display when entered by the user.
20 KEY 1,""	Would erase Soft Key 1.

The following routine initializes the first 5 soft keys:

```
1 KEY OFF 'Turn off key display during init.
10 DATA KEY1,KEY2,KEY3,KEY4,KEY5
20 FOR I=1 TO 5:READ SOFTKEYS$(I)
30 KEY I,SOFTKEYS$(I)
40 NEXT I
50 KEY ON 'now display new softkeys.
```

1.2.10 The Command Super Shift Keys

BASIC supports a Command Super Shift which enables easy entry of BASIC statement Keywords.

A Basic Keyword is entered when the Command Super Shift (or ALT) key is held down while one of the alphabetic keys A - Z is depressed. Keywords associated with each are summarized below. Keys not having reserved words are noted by ****.

A - AUTO	N - NEXT
B - BSAVE	O - OPEN
C - COLOR	P - PRINT
D - DELETE	Q - ****
E - ELSE	R - RUN
F - FOR	S - SCREEN
G - GOTO	T - THEN
H - HEX$	U - USING
I - INPUT	V - VAL
J - ****	W - WIDTH
K - KEY	X - XOR
L - LOCATE	Y - ****
M - MOTOR	Z - ****

1.2.11 The LOCATE Statement

Function:

The LOCATE statement moves the Cursor to the specified position on the active Screen. Optional parameters turn the blinking cursor on and off and define the start and stop raster lines for the cursor.

Format:

```
LOCATE [row] [, [col] [, [cursor] [, [start] [,stop] ]]]
```

WHERE:

row — Is the Screen Line number. A numeric expression returning an unsigned Integer in the range 1 to 24.

col — Is the Screen Column number. A numeric expression returning an unsigned Integer in the range 1 to 40, 1 to 80, or 1 to 132, depending upon Screen Width.

cursor — Is a boolean value indicating whether the cursor is visible or not. 0 for off, non-zero for on.

start — Is the cursor starting scan line. A numeric expression returning an unsigned Integer in the range 0 to 31.

stop — Is the cursor stop scan line. A numeric expression returning an unsigned Integer in the range 0 to 31.

Action:

Moves the cursor to the specified position. subsequent PRINT statements begin placing characters at this location. Optionally may be used to turn the blinking cursor on or off, or change the size of the blinking cursor.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" Error. Previous values are retained.

2. Any parameter may be omitted. Omitted parameters assume the old value.

3. If the start scan line parameter is given and the stop scan line parameter is omitted, stop assumes the start value.

4. Cursor Blink is not selectable and always blinks 16 times a second.

5. The 25th line is reserved for Soft Key display and may not be written over, even if Soft Key display is Off.

Example:

10 LOCATE 1,1	Moves to the home position in the upper left hand corner.
20 LOCATE ,,1	Make the blinking cursor visible, position remains unchanged.
30 LOCATE ,,,7	Position, and cursor visibility remain unchanged. Set the cursor to display at the bottom of the character starting and ending on scan line 7.
40 LOCATE 5,1,1,0,7	Move to Line 5, column 1, turn cursor on, cursor will cover entire character cell starting at scan line 0 and ending on scan line 7.

1.2.12 The LIST Statement

Function:

The LIST statement allows a program to be listed to the Screen, Cassette, or Line Printer.

Format:

LIST [[<line no.> [- [<line no.>]]] [,<dev>]]

WHERE:

line no. Is a valid line number in the range 0 to 65529.

dev Is a valid string expression returning one of the following devices:

SCRN:, CAS1:, or LPT1:.

Rules:

1. If the optional device specification is omitted, the specified lines are listed to the screen.

2. Listings directed to the screen by omitting the device specifier may be stopped at any time by depressing BREAK. Listings directed to devices may not be interupted, and will list until the range is exhausted. That is; LIST <range> may be interrupted, but LIST <range> ,"SCRN:" may not.

3. If the line range is omitted, the entire program is listed.

4. When the dash (-) is used in a line range, three options are available:

   a. If only the first number is given, that line and all higher numbered lines are listed.

   b. If only the second number is given, all lines from the beginning of the program through the given line are listed.

   c. If both numbers are specified, the inclusive range is listed.

Example:

```
LIST ,"LPT1:"        List program to the Line Printer.

LIST 10-20           List lines 10 thru 20 to Screen.

LIST 10- ,"SCRN:"    List lines 10 thru last to Screen.

LIST -200,"CAS1:"    List first thru line 200 to Cassette.

LIST 35-60,"CAS1:"   List lines 35 thru 65 to Cassette.
```

1.2.13 The GRAPHIC Statement

Some OEM machines have two video boards: color and (black and white). The black and white video board should not be confused with the black and white mode which the color video board can be put into by disabling the color burst circuitry. The black and white board does not support graphics and will not be discussed. Note that the use of the extended character set with special drawing characters is not considered "graphics".

There are two graphics modes that the color video software supports:

Medium resolution (320 by 200) - 2 bits per pixel,

and;

High resolution (640 by 200) - 1 bit per pixel.

Selection is made via the SCREEN statement with a first argument of 1 for the medium resolution mode and 2 for the high resolution mode. The statements used for graphics in Level 1 GW-BASIC are:

```
PSET
PRESET
LINE
```

The only graphics function in Level 1 GW-BASIC is:

```
POINT
```

Medium resolution mode is often confusing because of its color features. The final color card allows the 4 actual values that can be stored for each of the screen pixels to be independently associated with a color. The actual pixel values are of course 0,1,2 and 3.

When storing into graphics memory with PSET, PRESET or LINE the user will select the "attribute" from one of these 4 values. This does not affect the current palette selection which determines the color actually associated with each of the four pixel values. Only the color statement affects the palette. When Basic puts characters up on the screen in medium resolution mode, the foreground attribute is always 3 and the background attribute is always 0.

The main source of confusion is that in alpha mode there is a direct association between an attribute and the color it displays. However, in medium resolution this association can be dynamically adjusted. This is an important feature as will be described in an example later.

The high resolution mode is fairly simply since a pixel only has a value of zero or one. A zero pixel is always associated with the color black. A one pixel can be associated with various intensities of white through the first argument to the color statement, but this is a feature of little value.

When characters are formed on the screen in High resolution mode they are always 1 in the foreground and 0 in the background. (White on Black, no blinking).

1.2.13.1 The PSET Statement

The simplest graphics statement is PSET. The syntax is:

```
PSET ( xcoordinate , ycoordinate ) [ , attribute ]
```

The first argument to PSET is the coordinate of the point to PSET. Coordinates always can come in one of two forms:

```
STEP ( x offset, y offset) or
( absolute x, absolute y)
```

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced. Examples are:

```
(10,10) absolute form
STEP (10,0) offset 10 in x and 0 in y
(0,0) origin
```

Note that when Basic scans coordinate values it will allow them to be beyond the edge of the screen, however values outside the integer range (-32768 to 32767) will cause an overflow error.

Note that (0,0) is always the upper left hand corner. It may seem strange to start numbering y at the top so the bottom left corner is (0,199) in both high-resolution and medium resolution, but this is standard.

PSET allows the attribute argument to be left off and it is defaulted to 3 in medium resolution and 1 in high resolution, since these are the foreground attributes for those modes.

Example:

```
10 FOR i=0 TO 100
20 PSET (i,i)
30 NEXT
(draw a diagonal line to (100,100))
40 FOR i=100 TO 0 STEP -1
50 PSET (i,i),0
60 NEXT
(clear out the line by setting each pixel to 0)
```

1.2.13.2 The PRESET Statement

PRESET has an identical syntax to PSET. The only difference is that if no third parameter is given the backround color - zero is selected. When a third argument is given, PRESET is identical to PSET.

Line 50 in the example above could be:

```
50 PRESET (i,i)
```

If an out of range coordinate is given to PSET or PRESET no action is taken nor is an error given. If an attribute greater than 4 is given this will result in illegal function call. Attribute value 2 will be treated like 0 in hi-res and 3 will be treated like 1 for compatibility with medium resolution.

1.2.13.3 The POINT Function

The point function allows the user to read the attribute value of a pixel from the screen. If the point given is out of range the value -1 is returned. In medium resolution valid returns are:0,1,2,3. In hi-resolution they are 0 and 1.

Example:

```
10 SCREEN 1
20 FOR C=0 TO 3
30 PSET (10,10),C
40 IF POINT(10,10)<>C THEN PRINT "Broken Basic!"
50 NEXT C

5 SCREEN 2
10 IF POINT(i,i)<>0 THEN PRESET (i,i) ELSE PSET (i,i)
' invert current state of a point
20 PSET (i,i),1-POINT(i,i) 'another way to invert a point!
```

1.2.13.4 The LINE Statement

LINE is the most powerful of the graphics statement. It allows a group of pixels to be controlled with a single statement.

Syntax:

```
LINE [(xl,yl)] -(x2,y2) [[,[attribute][,b[f]]
```

The simplest form of line is:

```
LINE -(x2,y2)
```

This will draw from the last point to the point (x2,y2) in the foreground attribute.

We can include a starting point also:

```
LINE (0,0)-(319,199) 'draw diagonal line down screen
LINE (0,100)-(319,100) 'draw bar across screen
```

We can append the attribute to draw the line in:

```
LINE (10,10)-(20,20),2   'draw in color 2!
```

```
10 CLS
20 LINE -(rnd*319,rnd*199),rnd*4
30 GO TO 20
'draw lines forever using random attribute --
(should look nice when color card works)
```

```
10 FOR x=0 TO 319
20 LINE (x,0)-(x,199),x AND 1
30 NEXT
(draw alternating pattern - line on line off)
```

The final argument to line is ",b" -- box or ",bf" filled box. The syntax indicates we can leave out the attribute argument and include the final argument as follows:

```
LINE (0,0)-(100,100),,b   'draw box in foreground
```

or include it:

```
LINE (0,0)-(200,200),2,bf  'filled box attribute 2
```

The ",b" tells basic to draw a rectangle with the points (xl,yl) and (x2,y2) as opposite corners. This avoids giving the four LINE commands:

```
LINE (xl,yl)-(x2,yl)
LINE (xl,yl)-(xl,y2)
LINE (x2,yl)-(x2,y2)
LINE (xl,y2)-(x2,y2)
```

which perform the equivalent function.

The ",bf" means draw the same rectangle as ",b" but also fill in the interior points with the selected attribute.

When out of range coordinates are given the line command the coordinate which is out of range is given the closest legal value. In other words, negative values become zero, y values greater than 199 become 199 and x values greater than 319 in medium res become 319 and greater than 639 in hi resolution become 639.

In the examples and syntax the coordinate form STEP(xoffset,yoffset) is not shown. However this form can be used whereever a coordinate is used. Note that all of the graphics statements and functions update the "more recent point used". In a line command if the relative form is used on the second coordinate it is relative to the first coordinate. The only other way "the most recently used" point is changed is that SCREEN and CLS initialize it to be the point in the middle of the screen (160,100) for medium and (320,100) for hi resolution.

The graphics commands have been fully optimized to take advantage of the 8088. They are significantly faster than other machines.

Last Example:

```
10 CLS
20 LINE -(rnd*639,rnd*199),rnd*2,bf
30 GO TO 20
```

1.2.14 The SCREEN Statement

Function:

The SCREEN statement sets the screen attributes.

Format:

SCREEN [mode] [, [burst] [, [apage] [,vpage]]]

WHERE:

mode Is a valid numeric expression returning an unsigned Integer value

0	80x25 Alphanumeric mode
1	320x200 Graphics mode
2	640x200 Graphics mode
3	40x16
4	80x16 Alphanumeric mode
5	132x16 Alphanumeric mode
6	132x25 Alphanumeric mode
7	512x254 Graphics mode
8	1024x254 Graphics mode
9	320x254 Graphics mode

burst A valid numeric expression returning a boolean result. This parameter is not used, but is included for future versions.

apage Active page. Valid in alpha only. A numeric expression returning an unsigned Integer in the range 0 to 7. Selects the page to be written to.

vpage Visual page. Valid in alpha only. Same values as apage above, selects which page is to be displayed on the screen. May be different than the active page.

Action:

If all parameters are legal, the new screen mode is stored, the screen is erased, Foreground color is set to white, and background and border colors are set to black.

If the new screen mode is the same as the previous mode, nothing is changed.

If the mode is Alpha, and only xapagex and xvpagex are specified, the effect is that of changing display pages for viewing.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" error. Previous values are retained.

2. Any parameter may be omitted. Omitted parameters assume their old values.

Example:

```
10 SCREEN ,,1,2          'Mode and Color burst
                          unchanged,
                         'use active page 1,
                         'display page 2.
20 SCREEN 2              'Switch to high res graphic
                          mode.
30 SCREEN 1,1            'Switch to medium res Color
                          Graphics.
40 SCREEN ,0             'Medium res graphics, color
                          off.
```

1.2.14.1 The SCREEN Function

Function:

The SCREEN Function returns the ordinal of the character from the screen at the specified row (line) and column.

Format:

```
x = SCREEN(row,col [,z])
```

WHERE:

x is a numeric variable receiving the ordinal returned.

row is a valid numeric expression returning an unsigned Integer from 1 to the maximum number of rows for the current mode.

col is a valid numeric expression returning an unsigned Integer in the range 1 to the maximum screen width.

z is a valid numeric expression returning a boolean result.

Action:

The ordinal of the character at the specified coordinates is stored in the numeric variable. If the optional parameter <z> is given and non-zero, the color attribute for the character is returned instead.

Rules:

1. Any values entered outside of these ranges will result in an "Illegal Function Call" error.

Example:

```
100 X = SCREEN (10,10)      'If the character at 10,10 is
                             A, then return 65.

110 X = SCREEN (1,1,1)      'Return the color attribute
                             of the character in the
                             upper left hand corner of
                             the screen.
```

1.2.15 The WIDTH Statement

Function:

The WIDTH statement sets the printed line width in number of characters for the Screen and Line Printer.

Format:

```
WIDTH <size>
WIDTH <file no.>,<size>
WIDTH <dev>,<size>
```

WHERE:

size — A valid numeric expression returning an integer result in the range 0 to 255. This is the new width.

file no. — A valid numeric expression returning an integer in the range 1 to 4. This is the number of the file OPENed.

dev — A valid string expression returning the device identifier. Valid devices are SCRN: and LPT1:.

Action:

Depending upon the device specified, the following actions are possible:

```
WIDTH <size>
        or
WIDTH "SCRN:",<size>
```

```
WIDTH "LPT1:",<size>
```

Used as a deferred width assignment for the Line Printer. This form of width stores the new width value without actually changing the current width setting. A subsequent OPEN "LPT1:" FOR OUTPUT AS <number> will use this value for width while the file is open.

```
WIDTH <file no.>,<size>
```

If the file is open to LPT1:, the Line Printer's width is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1: in the Level 1 GW-BASIC.

Rules:

1. Valid width for the Line Printer is 1 to 255.

   Any value entered outside of this range will result in an "Illegal Function Call" error. The previous value is retained.

2. Width has no effect on the Keyboard (KYBD:).

3. Maximum EPSON Line Printer Width is 80. However, WIDTH does not complain about values between 80 and 255.

4. Specifying WIDTH 255 for the Line Printer (LPT1:) disables line folding. This has the effect of infinite width.

5. On the DOT, the WIDTH statement sets the CRT mode to the first display mode that has xsizex or more columns.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
     .
     .
     .
6020 WIDTH #1,40
```

In the preceeding example, line 10 stores a Line Printer width of 75 characters per line.

Line 20 opens file #1 to the Line Printer and sets the width to 75 for subsiquent PRINT #1,... statements.

Line 6020 changes the current Line Printer width to 40 characters per line.

```
SCREEN 1,1      Set Screen to Medium res. Color Graphics.
WIDTH 80        Change Screen to Hi-res. Graphics.
WIDTH 40        Changes Screen back to Medium res.

SCREEN 0,1      Changes Screen to 40x25 Alpha Color Mode.
WIDTH 80        Changes Screen to 80x25 Alpha Color Mode.
```

1.3 LEVEL 2 / GW-BASIC INITIALIZATION

1.3.1 Level 2 Initialization

Level 2 GW-BASIC is loaded and executed by typing:
L2BASIC in response to the MS-DOS command line prompt; A>

Upon loading, Basic responds with the banner:

```
<OEM specific Product Identification>
Version x.xx Copyright XXXX 1982.
nnnnn Bytes Free

Ok
```

The Basic operating environment may be altered somewhat by specifying option switches following L2BASIC on the command line. The format of L2BASIC's command line is:

```
L2BASIC [<filename>] [/F:<number of files>] [/S:<lrecl>]
     [/C:<buffer size>] [/M:<highest memory location]
```

WHERE:

filename

Is the file name of a Basic Program. If <filename> is present, Basic proceeds as if a RUN <filename> command were given after initialization is complete. A default file extention of .BAS is assumed if none is given. This allows Basic programs to be run in batch by putting this form of the command line in an AUTOEXEC.BAT file. Programs run in this manner will need to exit via the SYSTEM in order to allow the next command from the AUTOEXEC.BAT file to be executed.

/F:<number of files>

If present, sets the maximum number of files that may be open simultaneously during the execution of a Basic program. Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size may be altered via the /S: option switch. If the /F: option is omitted, the number of files defaults to 3.

/S:<lrecl>

If present, sets the maximum record size allowed for use with random files. NOTE: the record size option to the OPEN statement cannot exceed this value. If the /S: option is omitted, the record size defaults to 128 bytes.

/C:<buffer size>

If present, controls RS232 Communications. If RS232 cards are present, /C:0 disables RS232 support. Any subsequent I/O attempts will result in a "Device Unavailable" error. Specifying /C:<n> allocates <n> bytes for the receive buffer and 128 bytes for the transmit buffer for each RS232 card present. If the /C: option is omitted, 256 bytes are allocated for the receive buffer and 128 bytes for the transmit buffer of each card present. The /C: switch has no affect when RS232 cards are not present.

/M:<highest memory location>

When present, sets the highest memory location that will be used by Basic. Basic will attempt to allocate 65k of memory for the Data and Stack segments. If machine language subroutines are to be used with Basic programs, use the /M: switch to reserve enough memory for them.

NOTE: <number of files>, <lrecl>, <buffer size>, and <highest memory location> are numbers that may be Decimal, Octal (preceeded by &O), or Hexadecimal (preceeded by &H).

Example:

A>L2BASIC PAYROLL	Use all of memory and 3 files, load and execute PAYROLL.BAS.
A>L2BASIC INVENT/F:6	Use all of memory and 6 files, load and execute INVENT.BAS.
A>L2BASIC /C:0/M:32768	Disable RS232 support and use only the first 32k of memory.
A>L2BASIC /F:4/S:512	Use 4 files and allow a maximum record length of 512 bytes.
A>L2BASIC TTY/C:512	Use all of memory and 3 files, Allocate 512 byte to RS232 receive buffers and 128 bytes to transmit buffers, load and execute TTY.BAS.

1.3.2 Error Messages

The following error messages are provided in Level 2 Basic in addition to those found in Level 1 Basic:

68 Device Unavailable
An attempt was made to open a file to a non-existent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:, or was disabled by the user. This occurs if an OPEN "COM1:... statement is executed but the user disabled RS232 support via the /C:0 switch directive on the command line.

69 Communication buffer overflow
Occurs when a communication input statement is executed but the input queue was already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs' will attempt to clear this fault unless characters continue to be received faster than the program can process them. In this case several options are available:

1. Increase the size of the COM receive buffer via the /C: switch.

2. Implement a "hand-shaking" protocol with the host/satellite such as XON/XOFF as demonstrated in the TTY programming example to turn transmit off long enough to catch up.

3. Use a lower Baud rate for transmit and receive.

70 Disk Write Protect
This is one of 3 "hard" disk errors returned from the diskette controller. This occurs when an attempt is made to write to a diskette that is write protected. Use an ON ERROR GOTO statement to detect this situation and request user action.

Other possible "hard" disk errors are:

71 Disk not Ready
Occurs when the diskette drive door is open or a diskette is not in the drive. Again use an ON ERROR GOTO statement to recover.

72 Disk Media Error
Occurs when the FDC controller detects a hardware or media fault. This usually indicates harmed media. Copy any existing files to a new diskette and re-format the hurt diskette. FORMAT will flag the bad tracks and place them in a file "badtrack". The remainder of the diskette is now usable.

1.3.3 GW-BASIC Statement Extensions

1.3.3.1 The DATE$ Statement

Function:

The DATE$ statement may be used to set or retrieve the current date.

Format:

```
DATE$ = <string expr>     To set the current date.
<string expr> = DATE$     To get the current date.
```

WHERE:

string expr — Is a valid string literal or variable.

Action:

The current date is fetched and assigned to the string variable if DATE$ is the expression in a LET or PRINT statement.

The current date is stored if DATE$ is the target of a string assignment.

Rules:

1. If <string expr> is not a valid string, a "Type mismatch" Error will result. Previous values are retained.

2. For <string var>= DATE$, DATE$ returns a 10 character string in the form "mm-dd-yyyy" where mm is the month (01 to 12), dd is the day (01 to 31) and yy is the year (1980 to 2099).

3. For DATE$ = <string expr>, <string expr> may be one of the following forms:

   "mm-dd-yy"
   "mm/dd/yy"
   "mm-dd-yyyy"
   or
   "mm/dd/yyyy"

   If any of the values are out of range or missing, an "Illegal Function Call" error is issued. Any previous date is retained.

Example:

```
DATE$ = "01-01-81"
Ok
PRINT DATE$
01-01-81
Ok
```

1.3.3.2 The TIME$ Statement

Function:

The TIME$ statement may be used to set or retrieve the current time.

Format:

```
TIME$ = <string expr>     To set the current time.
<string expr> = TIME$     To get the current time.
```

WHERE:

string expr Is a valid string literal or variable.

Action:

The current time is fetched and assigned to the string variable if TIME$ is the expression in a LET or PRINT statement.

The current time is stored if TIME$ is the target of a string assignment.

Rules:

1. If <string expr> is not a valid string, a "Type mismatch" Error will result.

2. For <string var>= TIME$, TIME$ returns an 8 character string in the form "hh:mm:ss" where hh is the hour (00 to 23), mm is the minutes (00 to 59), and ss is the seconds (00 to 59).

3. For TIME$ = <string expr>, <string expr> may be one of the following forms:

   a. "hh" Set the hour. Minutes and seconds default to 00.

   b. "hh:mm" Set the hour and minutes. Seconds default to 00.

   c. "hh:mm:ss" Sets the hour, minutes and seconds.

   If any of the values are out of range, an "Illegal Function Call" error is issued. The previous time is retained.

Example:

```
TIME$ = "08:00"
Ok
PRINT TIME$
08:00:04
Ok
```

The following program displays the current date and time on the 25th line of the screen and will "chime" on the hour in the manner broadcast by WWV.

```
10 KEY OFF:SCREEN 0:WIDTH 40:CLS
20 LOCATE 25,5
30 PRINT DATE$,,TIME$
40 SEC = VAL(MID$(TIME$,7,2))
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC
60 IF SEC = 0  THEN 1010
70 IF SEC = 30 THEN 1020
80 IF SEC < 57 THEN 20

1000 SOUND 1000,2:GOTO 20
1010 SOUND 2000,8:GOTO 20
1020 SOUND 400,4 :GOTO 20
```

1.3.3.3 The OPEN Statement

NOTE: Refer to the Microsoft Basic Manual for syntax of the "Old" OPEN syntax.

Function:

The OPEN statement establishes addressability between a physical device and an I/O buffer in the data pool.

Format:

```
OPEN [<dev>] <filename> [FOR <mode>] AS [#]<file number> [LEN=<
```

WHERE:

is optionally part of the filename string and may be one of the following:

- A: - D: for Disk
- KYBD: - Keyboard (input only)
- SCRN: - Screen (output only)
- LPT1: - Line Printer 1
- LPT2: - Line Printer 2
- LPT3: - Line Printer 3
- COM1: - RS232 Communications 1
- COM2: - RS232 Communications 2

filename — Is a valid string literal o. variable optionally containing a <dev>. If <dev> is omitted, disk A: is assumed. Refer to "DISK FILES" for naming conventions.

mode — determines the initial positioning within the file and the action to be taken if the file does not exist. The valid modes and actions taken are:

- **INPUT** — Position to the beginning of an existing file. A "File not found" error is given if the file does not exist.
- **OUTPUT** — Position to the beginning of the file. If the file does not exist, one is created.

APPEND — Position to the end of the file. If the file does not exist, one is created.

If the FOR <mode> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created. This is the Random I/O mode. That is, records may be read or written at will at any position within the file.

file number — is an integer expression returning a number in the range 1 thru 15. The number is used to associate an I/O buffer with a disk file or device. This association exists until a CLOSE or CLOSE <file number> statement is executed.

lrecl — is an integer expression in the range 2 to 32768. this value sets the record length to be used for random files (see the FIELD statement). If omitted, the record length defaults to 128 byte records.

Action:

For each device, the following OPEN modes are allowed:

```
KYBD: - INPUT only
SCRN: - OUTPUT only
LPT1: - OUTPUT only
LPT2: - OUTPUT only
LPT3: - OUTPUT only
COM1: - INPUT, OUTPUT, or random only
COM2: - INPUT, OUTPUT, or random only
Disk files allow all modes.
```

When a disk file is OPENed FOR APPEND, the position is initially at the end of the file and the record number is set to the last record of the file (LOF(x)/128). PRINT, WRITE, or PUT will then extend the file. The program may position elsewhere in the file with a GET statement. If this is done, the mode is changed to random and the position moves to the record indicated.

Once the position is moved from the end of the file, additional records may be appended to the file by executing a GET #x,LOF(x)/<lrecl> statement. This positions the file pointer at the end of the file in preparation for appending.

Rules:

1. Any values entered outside of the ranges given will result in an "Illegal Function Call" Error. The file is not opened.

2. If the file is opened as INPUT, attempts to write to the file will result in a "Bad File Mode" error.

3. If the file is opened as OUTPUT, attempts to read the file will result in a "Bad File Mode" error.

4. At any one time, it is possible to have a particular disk filename OPEN under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, to use different file numbers for different modes of access. Each file number has a different buffer, so several records from the same file may be kept in memory for quick access.

   A file may NOT be opened FOR OUTPUT, however, on more than one file number at a time.

5. If the LEN=<lrecl> option is used, lrecl may not exceed the value set by the /S:<lrecl> switch option to the command line.

Example:

```
10 OPEN "PARTS.DAT" AS #1 'for random I/O on Disk A:

10 OPEN "KYBD:" FOR INPUT AS #2

10 OPEN "B.INVENT.DAT" FOR APPEND AS #1
```

1.3.4 Disk Files

Disk filenames follow the normal XXX DOS naming convention. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise drive A: is assumed. A default extention of .BAS is used on LOAD, SAVE, MERGE, RUN and CHAIN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

1.3.4.1 The LOF Function

Function:

The LOF function returns the number of bytes allocated to the file.

Format:

LOF(<file number>)

WHERE:

file number is tied to a currently open file.

Example:

```
10 OPEN "FILE.BIG" AS #1
20 GET #1,LOF(1)/128
     .
     .
     .
```

The above statements will get the last record of the file FILE.BIG assuming that the file was created with a default record length of 128 bytes.

1.3.4.2 The VARPTR Function

Function:

For files, the VARPTR function returns the address of the first byte of the File Control Block (FCB) for the opened file.

Format:

VARPTR(<file number>)

WHERE:

file number is tied to a currently open file. Offsets to information in the FCB from the address returned by VARPTR are:

OFF	SIZE		CONTENTS
0	1	Mode	The mode in which the file was opened: 1 - Input Only 2 - Output Only 4 - Random I/O 16 - Append Only 32 - Internal use 64 - Future use 128 Internal use
1	38	FCB	Disk File Control Block. Refer to DOS User's Guide for contents.
39	2	CURLOC	Number of sectors read or written for sequential access. For random access, it contains the last record number +1 read or written.
41	1	ORNOFS	Number of bytes in sector when read or written.
42	1	NMLOFS	Number of bytes left in Input buffer.
43	3	***	Reserved for future expansion.

46	1	DEVICE	Device Number: 0-9 - Disks A: through J: 255 - KYBD: 254 - SCRN: 253 - LPT1: 251 - COM1: 250 - COM2: 249 - LPT2: 248 - LPT3:
47	1	WIDTH	Device Width.
48	1	POS	Position in buffer for PRINT.
49	1	FLAGS	Internal use during LOAD/SAVE not used for data files.
50	1	OUTPOS	Output Position used during tab expansion.
51	128	BUFFER	Physical Data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in Sequential I/O mode.
179	2	VRECL	Variable length record size. Default is 128. Set by length option in OPEN statement.
181	2	PHYREC	Current physical record number.
183	2	LOGREC	Current logical record number.
185	1	***	Future use.
186	2	OUTPOS	Disk files only. Output position for PRINT, INPUT and WRITE.
188	<n>	FIELD	Actual FIELD data buffer. Size is determined by /S: switch. VRECL bytes are transferred between BUFFER and FIELD on I/O operations. Use this offset to examine File data in Random I/O mode.

Example:

```
10 OPEN "DATA.FIL" AS #1
20 FCBADR = VARPTR(#1) 'FCBADR contains start of FCB
30 DATADR = FCBADR+188  'DATADR contains address of data buffer
40 A$ = PEEK(DATADR)    'A$ contains lst byte in data buffer.
```

1.3.5 The RS232 Communication Option

This section describes the Basic statements required to support RS232 Asynchronous communication with other computers and peripherals.

1.3.5.1 OPENing a COM File

Function:

OPEN "COM..." allocates a buffer for I/O in the same fashion as OPEN for disk files.

Format:

OPEN "DEV:<speed>,<parity>,<data>,<stop>" AS [#]<file number>

WHERE:

DEV: Is a valid communications device. Valid devices are:
COM1: and COM2:

speed Is a literal integer specifying the transmit/receive baud rate. Valid speeds are: 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, 9600.

parity Is a one character literal specifying the Parity for Transmit and Receive as follows:

- S SPACE, Parity bit always transmitted and received as space (0 bit).
- O ODD, Odd Transmit/Receive Parity checking.
- M MARK, Parity bit always transmitted and received as mark (1 bit).
- E EVEN, Even Transmit/Receive Parity checking.
- N NONE, No Transmit Parity, No Receive Parity checking.

data Is a literal integer indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8.

NOTE: 4 data bits with no parity is illegal. Also 8 data bits with any parity is illegal.

stop Is a literal integer indicating the number of stop bits. Valid values are: 1 or 2. If omitted then 75 and 110 bps transmit two stop bits, all others transmit one stop bit.

file number Is an integer expression returning a valid file number. The number is then associated with the file for as long as it is OPEN and is used to refer other COM I/O statements to the file.

DEFAULTS: Missing parameters invoke the following defaults:
speed - 300 bps, parity - EVEN, bits - 7.

NOTE: A COM device may be OPENed to only one file number at a time.

Possible Errors:

Any coding errors within the File Name String will result in a "Bad File Name" Error. An indication as to which parameter is in error will not be given.

A "Device Timeout" error will occur if Data Set Ready (DSR) is not detected. Refer to hardware documentation for proper cableing instructions.

Example:

```
10 OPEN "COM1: " AS 1
```

File 1 is opened for communication with all defaults. Speed at 300 bps, Even Parity, and 7 data bits, one stop bit.

```
20 OPEN "COM1:2400 " AS #2
```

File 2 is opened for communication at 2400 bps. Parity and number of data bits are defaulted.

```
10 OPEN "COM1:1200,N,8" AS #1
```

File number 1 is opened for Asynchronous I/O at 1200 Bits/second, no parity is to be produced or checked, and 8 bit bytes will be sent and received.

1.3.5.2 Communication I/O

Since the communication port is opened as a file, all Input/Output statements that are valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files. They are: INPUT #<file number>, LINE INPUT #<file number>, and the INPUT$ function.

COM sequential output statements are the same as those for disk, and are: PRINT #<file number>, and PRINT #<file number> USING.

Refer to INPUT and PRINT sections for details of coding syntax and usage.

GET and PUT are only slightly different for COM files, see "The GET and PUT statements for COM files".

1.3.5.2.1 The COM I/O Functions

The most difficult aspect of asyncronous communication is being able to process characters as fast as they are received. At rates above 2400 bps., it is necessary to suspend character transmission from the host long enough to "catch up". This can be done by sending XOFF (ctrl S). to the host and XON (ctrl Q) when ready to resume.

Level 2 Basic provides three functions which help in determining when an "over-run" condition is eminent. These are:

LOC(x) Returns the number of characters in the input que waiting to be read. The input que can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the que, LOC(x) returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for the programmer to test for string size before reading data into it. If fewer than 255 characters remain in the que, LOC(x) returns the actual count.

LOF(x) Returns the amount of free space in the input que. That is, /C:<size>-LOC(x). Use of LOF may be used to detect when the input que is getting full. In practicality, LOC is adaquate for this purpose as will be demonstrated in the programming example.

EOF(x) If true (-1), indicates that the input que is empty. Returns false (0) if any characters are waiting to be read.

Possible Errors:

1. "Communication Buffer Overflow" If a read is attempted after the input que is full, (i.e. LOC(x) returns 0).

2. "Device I/O Error" If any of the following line conditions are detected on receive; Overrun Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.

3. "Device Fault" If Data Set Ready (DSR) is lost during I/O.

1.3.5.2.2 The INPUT$ Function

The INPUT$ function is preferred over the INPUT and LINE INPUT statements when reading COM files, since all Ascii characters may be significant in communications. INPUT is least desirable because input stops when a comma (,) or carriage-return is seen and LINE INPUT terminates when a carriage-return is seen.

INPUT$ allows all characters read to be assigned to a string. Recall from the rules for coding that INPUT$ will return X characters from the #Y file. The following statements then are most efficient for reading a COM file:

```
10 WHILE NOT EOF(1)
20   A$=INPUT$(LOC(1),#1)
30 ...
40 ... Process data returned in A$ ...
50 ...
60 WEND
```

The above sequence of statements read: ".. While there is something in the input que, return the number of characters in the que and store them in A$. If there are more than 255 characters, only 255 will be returned at a time to prevent "String Overflow". Futher, if this is the case, EOF(1) is false and input continues until the input que is empty.." Simple, concise, and fast.

1.3.5.2.3 The GET and PUT Statements

Function:

GET and PUT allow fixed length I/O for COM.

Format:

```
GET <file number>,<nbytes>
PUT <file number>,<nbytes>
```

WHERE:

file number — Is an integer expression returning a valid file number.

nbytes — Is an integer expression returning the number of bytes to be transferred into or out of the file buffer. nbytes cannot exceed the value set by the /S: switch when Basic was invoked.

Because of the low performance associated with telephone line communication, it is recommended that GET and PUT not be used in such applications.

Examples:

The following program (next page) enables the GW Machine to be used as a conventional terminal. Besides Full Duplex communication with a host, the TTY program allows data to be "Down-loaded" to a file. Conversly, a file may be "Up-loaded" (transmitted) to another machine.

In addition to demonstrating the elements of Asyncronous communication, this program should be useful in transfering Basic programs and Data to and from the GW machine.

NOTE: this program is set up to communicate with Microsoft's DEC-20, that is, the use of XON and XOFF. you may want to modify it further for your environment.

1.3.6 Notes on the TTY Programming Example

Line No.	Comments.
10	Sets the SCREEN to Black and White Alpha Mode and sets the Width to 80.
15	Turns off the Soft Key Display, Clears the screen, and makes sure that all files are closed.
NOTE:	Asynchronous implies character I/O as opposed to line or Block I/O. Therefore, all PRINT's (either to the COM file or screen) are terminated with a semi-colon (;). This retards the carriage-return line-feed normally issued at the end of a PRINT statement.
20	Define all numeric variables as INTEGER. Primarily for the benefit of the subroutine at 600-620. Any program looking for speed optimization should use integer counters in loops where possible.
25-30	Clears the 25th line starting at column 1.
40	Define Boolean TRUE and FALSE.
50	Define the Ascii (ASC) value of the MENU Key.
60	Defines the Ascii XON, XOFF characters.
100-130	Prints program-id and asks for baud rate (speed). Opens Communications to file number 1, Even parity, 7 data bits.

Programmer exercise:

Modify this section to check for valid baud rates before continuing.

200-280 This section performs Full Duplex I/O between the Video Screen and the device connected to the RS232 connector as follows:

1. Read a character from the keyboard into A$. Note that INKEY$ returns a null string if no character is waiting.

2. If no character is waiting then go see if any characters are being received. If a character is waiting at the keyboard then:

3. If the character was the MENU Key, then the user is ready to download a file, so go get file name.

4. If character (A$) is not the MENU key then send it by writing to the communication file (PRINT #1...).

5. At 230 see if any characters are waiting in COM buffer. If not, then go back and check keyboard.

6. At 240, if more than 128 characters are waiting then, set PAUSE flag saying we are suspending input and send XOFF to host stopping further transmission.

7. At 250-260, read and display contents of COM buffer on screen until empty. Continue to monitor size of COM buffer (in 240). Suspend transmission if we fall behind.

8. Finally, resume host transmission by sending XON only if suspended by previous XOFF. Repeat process until MENU Key struck.

300-320 Get Disk file name we are down-loading to. Open the file to tie number 2.

400-420 Asks if file named is to be transmitted (Upload) or received (Down-loaded).

430 Sends a carriage-return to the host to begin the down-load. This program assumes that the last command sent to the host was to begin such a transfer and was missing only the terminating carriage-return. If a DEC System is the host, then such a command might be:

```
COPY TTY:=MANUAL.MEM<MENU Key>
```

WHERE: The MENU Key was struck instead of RETURN.

500 When no more characters are being received (LOC(x) returns 0), then perform a timeout routine (explained later).

510 Again, if more than 128 characters are waiting, signal a pause and send XOFF to the host while we catch-up.

520-530 Read all character in COM que (LOC(x)) and write them to disk (PRINT #2..) until we are caught up.

540-550 If a pause was issued, restart host by sending XON and clear the pause flag. Continue process until no characters are received for a pre-determined time.

600-650 This is the time-out subroutine. The FOR loop count was determined by experimentation. In short, if no character is received from the host for 17-20 seconds, then transmission is assumed complete. If any character is received during this time (line 610) then set I well above FOR loop range to exit loop and then return to caller. If host transmission is complete, close the disk file and return to being a terminal.

1000-1060 Transmit routine. Until end of disk file do:

Read one character into A$ with INPUT$ statement. Send character to COM device in 1020. Send a control-Z at end of file in 1040 in case receiving device needs one to close its file. Finally, in lines 1050 and 1060, Close our disk file, print completion message and go back to conversation mode in line 200.

9999 Presently not executed. As an exercise, add some lines to the routine 400-420 to exit the program via line 9999. This line closes the COM file left open and restores the Soft Key Display.

1.4 GW-BASIC / LEVEL 3

1.4.1 Level 3 Overview

Level 3 is the most advanced form of GW-BASIC available. It is a superset of Level 2 and Level 1 GW_BASIC and requires a disk based machine with 10k ram for MS-DOS and 24k for the Basic itself. Only the additional features available in Level 3 Basic will be described here. The only "overlap" in features between the lower levels and Level 3 is that an additional meaning is given to "GET" and "PUT" with Level 3.

In Level 2 these commands are used for random access I/O. In Level 3 they are used for disk I/O and graphics -- the distinction is made by examining the syntax. Specifically disk GET and PUT do not have a "@", STEP or "(" as their first character while graphics GET and PUT do.

Features originally defined for Level 3 GW_BASIC have been moved into other levels, specifically:

Level 1:
Soft key support, Super-Shift key support

Level 2:
TIME$, DATE$ and RS-232 support

The key Level 3 features are:

1. "Event" Trapping

   This allows a program to trap the occurance of a specific event by trapping to a specific program line. Events include: COM activity, Special key activity, Trigger activity and Light Pen activity.

2. Advanced Graphics

   CIRCLE, PUT, GET, PAINT, DRAW. these operations extend the ease and power of using the color video card graphics feature quite substantially. In particular, PAINT and DRAW allow efficient manipulaton of complex scene data. PUT and GET allow animation type operations.

3. Advanced Music Support

The PLAY statement allows advanced and easy usage of the planar speaker capability. Plug in music card support is not provided since this option has yet to be fully defined. This limits music to a single voice output.

NOTE: Several of the more powerful Level 3 features are among those excluded from support by the Basic compiler. Event Trapping and the "x" subcommand in the PLAY and DRAW statements are not available.

1.4.2 Event Trapping

The following are defined as "event specifiers":

COM (n) where n is the number of the COM channel (1 or 2)

KEY (n) where n is a function KEY number (1-14). 1 through 10 are the Softkeys F1 through F10. 11 through 14 are the cursor direction keys as follows: 11-Up, 12-Left, 13-Right, 14-Down. A KEY(0) ON, OFF, or STOP enables, disables or stops all 14 key events.

PEN - since there is only one PEN no number is given

STRIG(n) where n is 0 or 2 (0 for first trigger 2 for second)

We add the following statements:

ON <event specifier> GOSUB <line number>

This sets up an event trap line number for the specified event. A <line number> of 0 disables trapping for this event.

<event specifier> ON
<event specifier> OFF
<event specifier> STOP

These statements control the activation/deactivation of event trapping. When an event is ON, if a non-zero line number is specified for the trap with an ON statement then every time Basic starts a new statement it will check to see if the specified event has occured (the trigger has gone down, a function key was struck, the pen went down, a com character has come in) and if so it will perform a GOSUB to the line specified in the ON statement.

When an event is OFF, no trapping takes place and the event is not remembered even if it takes place.

When an event is "stopped" no trapping can take place, but if the event happens this is remembered so an immediate trap will take place when an <event> ON is executed.

When a trap is made for a particular event the trap automatically causes a "stop" on that event so recursive traps can never take place the "return" from the trap routine automatically does an ON unless an explicit OFF has been performed inside the trap routine

When an error trap takes place this automatically disables all trapping.

Trapping will never take place when Basic is not executing a program.

RETURN <line number>

This optional form of RETURN in Level 3 is primarily intended for use with Event Trapping. The event trap routine may want to go back into the Basic program at a fixed line number while still eliminating the GOSUB entry that the trap created.

Use of the non-local RETURN must be done with care! Any other GOSUB, WHILE or FOR that was active at the time of the trap will remain active. If the trap comes out of a subroutine, any attempt to continue loops outside the subroutine will result in a "NEXT without FOR" error.

special notes about each type of trap:

KEY trapping

No type of trapping is activated when Basic is in direct mode. In particular, function keys resume their standard expansion meaning during input.

A key that causes a trap is not available for examination with the INPUT or INKEY$ statements so the trap routine for each key must be different if a different function is desired.

PEN trapping

PEN trapping "traps" downstrokes of the light pen. The only difference between Level 1 PEN ON and Level 3 PEN ON is if the ON PEN GOSUB statement has been executed with a non-zero line number. If so, in Level 3 Basic a downstroke will cause a trap. PEN(0) is not set for a downstroke that causes a trap.

COM trapping

Typically the COM trap routine will read an entire message from the COM port before returning back. It is not recommended to use the COM trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for COM to overflow.

STRIG trapping

Using STRIG(n) ON will activate the interrupt routine that checks the trigger status. Downstrokes that cause trapping will not set STRIG(0) or (2).

1.4.3 Advanced Graphics

Advanced Graphics extends the capabilities to manipulate the graphics mode bit map provided by the color video card. Once again we have the two modes: medium resolution (2 bits per pixel) and high resolution (1 bit per pixel).

Additional Statements:

CIRCLE (xcenter,ycenter),radius[,attribute[,start,end[,aspect]]]

The CIRCLE statement draws an ellipse with a center and radius as indicated by the first of its arguments. The default attribute is the foreground color; 3 in medium res, 1 in hi res. The start and end angle parameters are radian arguments between 0 and 2*PI which allow you to specify where drawing of the ellipse will begin and end. If the start or end angle is negative, the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (Note that this is different than adding 2*PI).

The aspect ratio describes the ratio of the X radius to the Y radius. The default aspect ratio is 5/6 in medium res and 5/12 in hi res, and will give a visual circle in either graphics mode assuming a standard monitor screen aspect ratio of 4/3.

If the aspect ratio is less than one, then the radius is given in X-pixels. If it is greater than one, the radius is given in Y-pixels. The standard relative notation may be used to specify the center point.

In many cases, an aspect ratio of 1 will give nicer looking circles in medium res mode. This will also cause the circle to be drawn somewhat faster. The start angle may be less than the end angle.

GET (x1,y1)-(x2,y2),array name
PUT (x1,y1),array[,action verb]

(where action verb is one of: PSET, PRESET, AND, OR, XOR)

The PUT and GET statements are used to transfer graphics images to and from the screen. PUT and GET make possible animation and high-speed object motion in either graphics mode.

The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The rectangle is defined the same way as the rectangle drawn by the LINE statement using the ",B" option.

The array is used simply as a place to hold the image and can be of any type except string. It must be dimensioned large enough to hold the entire image. The contents of the array after a GET will be meaningless when interpreted directly (unless the array is of type integer - see below)

The PUT statement transfers the image stored in the array onto the screen. The specified point is the coordinate of the top left corner of the image. An "Illegal Function call" error will result if the image to be transferred is too large to fit on the screen.

The action verb is used to interact the transferred image with the image already on the screen. PSET transfers the data onto the screen verbatum.

PRESET is the same as PSET except that a negative image (black on white) is produced.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode often used for animation. XOR causes the points on the screen to be INVERTED where a point exists in the array image. This behavior is exactly like the cursor on the screen. XOR has a uniqe property that makes it especially useful for animation: when an image is PUT against a complex background once TWICE, the background is restored unchanged. This allows you to move an object around the screen without obliterating the background.

The default action mode is XOR.

It is possible to GET an image in one mode and PUT it in another, although the effect may be quite strange because of the way points are represented in each mode.

In medium res mode, AND, OR and XOR have the following effects on color:

AND

array attrb	screen attrib 0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

OR

array attrb	screen attrib 0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

array attrb	screen attrib 0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Animation of an object is usually performed as outlined below:

1. PUT the object(s) on the screen.
2. Recalculate the new position of the object(s).
3. PUT the object(s) on the screen a second time at the old location(s) to remove the old image(s).
4. Go to step one, this time PUTting the object(s) at the new location.

Movement done this way will leave the background unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. The idea is to leave a border around the image when it is first gotten as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points. This method may be somewhat faster than the method using XOR described above since only one PUT is required to move an object (although you must PUT a larger image).

The storage format in the array is as follows:

```
2 bytes giving X dimension in BITS
2 bytes giving Y dimension
The array data itself
```

The data for each row of pixels is left justified on a byte boundary, so if there are less than a multiple of 8 bits stored, the rest of the byte will be filled out with zeros. The required array size in bytes is:

```
4+INT((x*bitsperpixel+7)/8)*y
```

WHERE: bitsperpixel is 2 in medium res and 1 in hi res.

The bytes per element of an array are:

```
2 for integer
4 for single precision
8 for double precision
```

An example:

Say you want to GET a 10 by 12 image into an integer array. The number of bytes required is 4+INT((10*2+7)/8)*12 or 40 bytes. So, you would need an integer array with at least 20 elements.

It is possible to examine the X and Y dimensions and even the data itself if an integer array is used. The X dimension is in element 0 of the array, and the Y dimension is found in element 1. It must be remembered, however, that integers are stored low byte first, then hi byte, but the data is transferred hi byte first (leftmost) and then lo byte.

PAINT (xstart,ystart)[,paint attribute[,border attribute]]

The PAINT statement will fill in an arbitrary graphics figure of the specified border attribute with the specified fill attribute. The paint attribute will default to the foreground attribute (3 or 1) if not given, and the border attribute defaults to the paint attribute.

For example, you might want to fill in a circle of attribute 1 with attribute 2. Visually, this could mean a ???? ball with a ???? border ((check these)).

Since there are only two attributes in hi res mode, this means whiteing out an area until white is hit, or blacking out an area until black is hit.

PAINT must start on a non-border point, otherwise PAINT will have no effect.

PAINT can fill any figure, but PAINTing "jagged" edges or very complex figures may result in an "Out of Memory" error. If this happens, you must use the CLEAR statement to increase the amount of stack space available.

DRAW <string expression>

The DRAW verb combines most of the capabilities of the other graphics statements into an easy-to-use object definition language called "Graphics Macro Language" (**). A GML command is a single character within a string, optionally followed by one or more arguments.

(**) Graphics Macro Language is a Trademark of Microsoft.

Movement commands:

Each of the following movement commands begin movement from the "current graphics position". This is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen (160,100 in medium res, 320,100 in hi res) when a program is RUN.

U [<n>]	Move up (scale factor * N) points
D [<n>]	Move down
L [<n>]	Move left
R [<n>]	Move right
E [<n>]	Move diagonally up and right
F [<N>]	Move diagonally up and left
G [<n>]	Move diagonally down and left
H [<n>]	Move diagonally down and right

The above commands move one unit if no argument is supplied.

M <X,Y> Move absolute or relative. If X is preceded by a "+" or "-", X and Y are added to the current graphics position, and connected with the current position with a line. Otherwise, a line is drawn to point X,Y from the current position.

The following prefix commands may precede any of the above movement commands:

B Move but don't plot any points.

N Move but return to original position when done.

A <n> Set angle n. n may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they will appear the same size as with 0 or 180 degrees on a monitor screen with the standard aspect ratio of 4/3.

C <n> Set attribute n. n may range from 0 to 3 in medium res, and 0 to 1 in hi res.

S <n> Set scale factor. n may range from 1 to 255. The scale factor multiplied by the distances given with U,D,L,R or relative M commands to get the actual distance traveled.

X <string> Execute substring (not supported by Basic compiler). This powerful command allows you to execute a second substring from a string, much like GOSUB in Basic. You can have one string execute another, which executes a third, and so on.

Numeric arguments can be constants like "123" or "=variable;", where variable is the name of a variable. (Not supported by Basic compiler).

1.4.4 PLAY <string expression>

PLAY implements a concept similar to DRAW by embedding a Music Macro Language (***) into the string data type.

The single character commands in play are:

A-G [#|+,-] Play the note. A "#" or "+" afterwards means sharp, and "-" means flat.

L <n> Length - Sets the length of each note. L4 is a quarter note, L1 is a whole note, etc. n may range from 1 to 64.

The length may also follow the note when it is desired to change the length only for the note. In this case, A16 is equivalent to L16A.

MF Music Foreground. Music (PLAY statement) and SOUND are to run in Foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. This is the initial default.

MB Music Background. Music (PLAY statement) and SOUND are to run in Background. That is, each note or sound is placed in a buffer allowing the Basic program to continue execution while music plays in the "background. Up to 32 notes (or rests) can be played in background at a time.

MN Music Normal. Each note will play 7/8ths of the time determined by L (length).

ML Music Legato. Each note will play the full period set by L (length).

MS Music Staccato. Each note will play 3/4ths of the time determined by L (length).

N <n> Play note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. N=0 means rest.

O <n> Octave - Sets the current octave. There are 7 octaves (0..6)

P <n> Pause. P may range from 1 to 64.

T <n> Tempo - Sets the number of L4's in a second. n may range from 32 to 255. Default is 120.

. Dot or Period. After each note causes the note to play 3/2 times the period determined by L (length) times T (tempo). Multiple dots may appear after the note. The period is scaled accordingly. (ex: A. 3/2, A.. 9/4, A... 27/8 etc). Dots may appear after a pause (P) and scale the pause length as described above.

X<string> Execute substring.

Because of the slow clock interrupt rate, some notes will not play at higher tempos; e.g. L64 at T255. Which note/tempo combinations these are must be determined through experimentation.

(***) Music Macro Language is a Trademark of Microsoft.

Computer Devices, Inc., 25 North Avenue, Burlington, Massachusetts 01803

0008-14662/3-83